### SER421: Web Applications and Mobile Systems Fall 2016

### Lab 1, due Thursday, 10/20/16 at 11:59pm via online submission to Blackboard

***THIS IS AN INDIVIDUAL LAB***

The goal Lab 1 is to get you "warmed up" for the coming semester by practicing skills in HTML5, Javascript, JSON, and HTTP. There are 3 activities to complete of varying degrees of complexity and point values.

**SUBMISSION INSTRUCTIONS (READ CAREFULLY and ASK QUESTIONS!):**

1. Create a zipfile name <asurite>\_421Lab1.zip where <asurite> is your ASURITE id. I would do kgary\_421Lab1.zip No RAR files!
2. The zipfile should have 3 files and 1 subdirectory named activity3. The 3 files in the top-level directory should be named README.txt, activity1.html, and activity2.docx. The activity3 folder will have your code for Activity 3.
3. The file named README.txt should have any information you want us to know. For example, if you only completed some of the parts of a multipart activity, you can indicate that and what is undone (we consider partial credit).
4. I strongly suggest, especially on programming problems, that you get a stable solution to a Part, save it, and then move on. We grade in parts – so for example in Activity 3 if you only get Parts A and B stable, give us that. You can also give us a partially done Part C which we will evaluate for partial credit, but your README.txt has to tell us that!

**Activity 1(22 points): Test the new HTML5 Input Types**

As we discussed in class, HTML5 has added a number of new input types. You can see a list at <http://www.w3schools.com/html/html_form_input_types.asp> in the middle of the page under "HTML5 Input Types".

*(6) PART A:* In an HTML document, create a single HTML form that includes each of these new HTML5 input types. Please put each input on its own line and have it labeled with the type name. For example, in class we used the form with firstname, lastname, email, and tel number with form elements one per line (newline with the <br/> tag). Do likewise here, e.g.

Color: \_\_\_\_\_\_\_\_\_\_\_

Date: \_\_\_\_\_\_\_\_\_\_\_

DateTime: \_\_\_\_\_\_\_\_\_\_\_

DateTime-local: \_\_\_\_\_\_\_\_\_\_\_

… and so on (ok that is a cheesy replica of an HTML form but you get the gist). W3schools lists 13 new input types.

The forms action be the echo URL <http://lead1.poly.asu.edu/cgi-bin/cgiquery.pl>, and the METHOD should be GET.

The name attribute of each input element should be <your asurite id>\_<typename>. For example, I would use kgary\_color, kgary\_date, kgary\_datetime, etc.

*(8) PART B:* Additionally, that same w3schools page summarizes input restrictions on form input types, and indicates new attributes min, max, pattern, required, and step. With these attributes, do the following:

1. Specify a min date of June 1, 2010 and a max date of October 1, 2016 on your Date input
2. Specify a min of -20 and a max 20 on your range input
3. Make sure all of the input elements must be filled in
4. Specify a default value of 10 in your number input

*(8) PART C:* Next, create a 14th input element of type "text" intended to hold the numbers you typically see at the bottom of a check (see <http://www.routingnumbers.org/>). Use the *pattern* attribute to create a custom validation for the combination of 9-digit routing number, 12-digit account number, and 4-digit check number. Constraints on the pattern:

1. All 3 numbers must be all digits
2. The routing number cannot start with 0
3. All 3 must be fixed length, padded with leading 0s if needed (account and check numbers) and have a single space between them.
4. The textbox must be lengthened to accommodate the entire string.
5. Label the area "Routing number" on your HTML form and name the input element "RTN"

**Activity 2 (18 points): HTTP Basics**

Download the "Tamper Data" plugin from https://addons.mozilla.org/en-US/firefox/addon/966. Run this tool and visit URLs:

1. http://www.cbsnews.com

2. https://search.yahoo.com/ (type in any search phrase and hit return)

For each URL, use the Tamper Data tool to find out the return values in the HTTP response header. Does the site set a cookie? Does the site use persistent connections? Does the site compress its content? Capture the response header pane in the lower right and paste it into your solution for each URL. You may put your answers in the same Word document as Activity 1. *I am OK if you want to use Firebug or Google Chrome Dev Tools instead of TamperData; I just find TD easier to follow personally*

**Activity 3 (60 points): Javascript and JSON**

For this activity you will create a Javascript-based postfix calculator. A postfix calculator is one where the operator appears at the end of the expression. Your expressions will be written in JSON.

*(15) PART A:* For the initial calculator, you only have to process add/subtract against a stored calculator value, initially 0. Example:

1. {"number" : 5, "operation" : "add"} returns 5 (assumes a starting init value of 0)
2. {"number" : 2, "operation" : "subtract"} returns 3 (5-2)
3. {"number" : 19, "operation" : "add"} returns 22 (19+3)

*(15) Part B:* Now add the ability to process nested expressions. For example (continuing the previous example)

1. {"expr": {"number" : 15, "operation" : "subtract"}, "operation" : "add"} returns 14 (22-15 = 7, then 7+7=14)
2. {"expr": {"expr": {"number" : 3, "operation" : "add"}, "operation" : "add"}, "operation" : "subtract"} returns 0 (14+3=17, 17+17=34, 34-34=0)

Expressions can be abitrarily nested

*(30) Part C:* OK if you think about Part B for a second, we wont get very meaningful expressions. So lets add a stack-based memory store (array implemented) to the calculator and new operations:

* Create a resizable array to hold the results of any calculation the user wants stored
* Create a new operation *push* that puts a value at the top of that stack
* Create a new operation *pop* that pops the top value off the stack and returns that value
* Create a *print* operation that writes out the contents of the stack from top to bottom on the screen
* The *subtract* and *add* operations do not change except that they use the top value on the stack *without popping it*. Further, neither operation *pushes the result*. So, all pushes and pops are done explicitly via the respective operations.

Lets retrace the example with these new features:

1. {"number" : 5, "operation" : "add"} returns 5 but does not store the 5 on the stack. The stack remains [0]
2. {"expr": {"number" : 2, "operation" : "subtract"}, "operation" : "push"} returns -2 and pushes -2 on top of the stack
3. {"expr": {"number" : 19, "operation" : "add"}, "operation" : "push"} returns 17 (-2+19) and pushes 17 to the top of the stack
4. {"operation" : "pop"} returns 17 and removes it from the stack
5. {"operation" : "print"} returns "-2 0"
6. {"expr": {"expr": {"operation" : "pop"}, "operation" : "add"}, "operation" : "push"} returns -2 (-2 + 0)
7. {"operation" : "print"} returns "-2 0"
8. {"operation" : "pop"} returns -2
9. {"operation" : "pop"} returns 0
10. {"operation" : "pop"} returns (what? You have an empty stack now)

Hints:

1. Create a simple grammar for the calculator
2. Think of your expressions as objects (because that is what they will be when you parse the JSON).
3. Then test in the interpreter what the structures are that you get back from JSON.parse
4. I used "add" and "subtract" on purpose instead of "+" and "-" so you can be more efficient
5. You do not have to check for well-formedness of the JSON and type errors (nagative testing). However you may have to type-check using "typeof", "instanceof" or other techniques to facilitate your processing.
6. Read up on the optional parameter of JSON.parse for one possible way to do your input checking.

Finally, notice I just gave you JSON strings above, I did not give you a function or method name. Depending on your design, you may implement either a function named *calc(json)* or an object named Pcalc with a method calc, e.g. pc = new PCalc(); pc.calc(*json*).