SER421 Lab 1 Fall 2020

THIS LAB IS TO BE COMPLETED INDIVIDUALLY

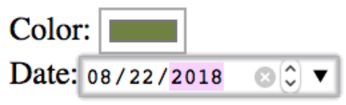
Learning Outcomes:

1. Gain proficiency in HTML5 semantic description tags and input validation
2. Gain proficiency in Javascript functional and object-oriented programming

Activity 1: HTML5 (35%)

As discussed in the videos, 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".

PART A (7): 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, one element per line (newline with the <br/> tag), e.g.



… and so on (that only shows the first 2 input types). W3schools lists 12 new input types.

The forms action can 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 <asurite>_<typename>. For example, for me it would be "kgary", I would use kgary_color, kgary_date, kgary_datetime_local, etc.

PART B (15): 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 January 1, 2010 and a max date of September 1, 2017 on your Date input
2. Specify a min of -10 and a max 10 on your range input
3. Make sure all of the input elements must be filled in
4. Specify a default value of 100 in your number input

PART C (13): Next, create a 13th 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 (an example is shown on that web page). Constraints on the pattern:

1. All 3 numbers must be all digits
2. All 3 must be fixed length, with leading 0s if needed to meet the fixed length, and have a single space between them.
3. The textbox must be lengthened to accommodate the entire string.
4. Label the area "Routing number" on your HTML form and name the input element "<asurite>_RTN" (e.g. kgary_RTN)

*NOTE: you are only **checking** for a valid routing number input, not trying to **fix** user input. This should be implemented by the HTML5 **pattern**, not by adding any Javascript (no Javascript allowed!).*

Save your HTML document solution as lab1act1.html. All parts can go in this file.

Grading rubrics for Activity 1:

Part A:

- *1 point deduction for any missing element types up to a max -4.*
- *1 point deduction for not using GET to the right form action.*
- *1 point deduction (total) if not naming the attributes properly.*
- *1 point deduction (at our discretion) for any other mistakes, such as not following naming instructions.*

Part B:

4 points each for #1-#3, 3 points for #4. Points awarded based on functional testing. Partial credit as follows:

- *1 point deduction on #1 or #2 for minor mistakes (typos)*
- *1 point deduction per each input element not filled in and not detected, up to a max of -4.*
- *Other partial credit at our discretion for approximation of correct answer (note, not a lot of wiggle room on this one!)*

Part C:

- *6 points for #1 (2 per each component of the RTN)*
- *5 points for #2, (1 for each component of fixed length: 9, 12, 4, and 1 each for a single space between components 1-2 & 2-3)*
- *2 points each for following instructions in #3-4. Partial credit for close answers.*

Activity 2: Javascript Programming (65%)
For this activity you will create a Javascript-based prefix calculator. A prefix calculator is one where the operator appears at the beginning of the expression. Your expressions will be written as JSON.

(15) PART A: For the initial calculator, process add/subtract against a single stored calculator value, initially 0. Examples:
1. `'{"op" : "add", "number" : 5}'` returns 5 (assumes a starting init value of 0)
2. `'{"op" : "subtract", "number" : 2}'` returns 3 (5-2)
3. `'{"op" : "add", "number" : 19}'` returns 22 (19+3)

You should implement a function *calc(string)* to compute your responses. Save your solution as lab1act2partA.js

(20) Part B: Now add the ability to process nested expressions. Examples (continuing the previous example)
4. `'{"op": "subtract", "expr" : {"op" : "add", "number" : 15}}'` returns 0 (22+15 = 37, then 37-37=0)
5. `'{"op": "add", "expr" : {"op" : "add", "expr" : {"op" : "subtract", "number" : 3}}}'` returns -12 (0-3=-3, -3+-3=-6, -6+-6=-12)

Note that these examples show that the calculator in this state is pretty useless – the value of the nested expression is treated as the "number", and is added/subtracted with the value in the calculator (itself), so you always get 0 or double the value!

Constraints:
1. Note the new key "expr" to denote the start of a nested expression. Expressions cannot be arbitrarily nested – meaning that you cannot have a JSON of the form: `{{<op><expr>}{<op><expr>}}`, rather it must always follow the pattern in #4 and #5.
2. Again, implement *calc(string)* to compute your responses

Save this as file lab1act2partB.js

(30) Part C: As discussed above in Part B, we don't get very meaningful expressions. So let's 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.

Let's retrace the example with these new features (assuming the stack starts with [0]):
1. `{"op" : "add", "number" : 5}` returns 5 (5+0) but does not store the 5 on the stack. The stack remains [0]
2. `{"op" : "push", "expr" : {"op" : "subtract", "number" : 2}}` returns -2 and pushes -2 on top of the stack [-2 0]
3. `{"op" : "push", "expr" : {"op" : "add", "number" : 19}}` returns 17 (-2+19) and pushes 17 to the top of the stack [17 -2 0]
4. `{"op" : "pop"}` returns 17 and removes it from the stack [-2 0]
5. `{"op" : "print"}` prints [-2 0]
6. `{"op" : "push", "expr" : {"op" : "add", "expr": {"op" :  "pop"}}}` returns -2 (-2 + 0) [-2 0]
7. `{"op" : "print"}` prints [-2 0]
8. `{"op" : "pop"}` returns -2 [0]
9. `{"op" : "pop"}` returns 0 []
10. `{"op" : "pop"}` returns (what? You have an empty stack now)

In order to store the stack, implement an *object of type PreCalc*. The object should take a number in the constructor and initialize the stack to that value. The stack should be stored as a property named *calcStack*, and the *calc* function should now be a method on the PreCalc object type. Save this as file lab1act2partC.js.

Hints:
1. Create a simple grammar for the JSON submitted to 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 (negative testing). However, you do have to test for proper state of the calculator (in particular, that last case where you pop an empty stack)

Submission:
- Put your 4 files together in a single zipfile named <asurite>_lab1.zip and submit via Canvas by the due date.
- You may (and probably should) include a README.txt in the root directory of the zipfile that explains to us anything you want us to know about your submission. We will read this before we start grading.
- You may include additional test case files if you develop them (you should!). Indicate in your README.txt.
- We reserve the right (at our discretion) to deduct points for poor code practices, such as a) not following instructions, and b) not following proper coding practices (documentation/comments, indentation, program structure, variable names, etc.)