Solutions to Quiz 1 (September 15)

Problem 1 (Scoping) (1 point).
Here’s a JavaScript program, where comments mark some positions where it would be legal to insert the keyword `var` before an assignment to variable name `x`.

```javascript
/* var? */ x = 7;

function f(y) {
    /* var? */ x = 20;

    var g = function(z) {
        /* var? */ x = z * 2;
        return x + 1;
    }

    var q = x * 7;
    if (x <= x) {
        /* var? */ x = g(y) % 4;
        return x * q;
    }
}
```

When we add all the `var` keywords from comments, how many more different run-time `x` variables are created in different scopes during execution, compared to when we leave out all the `var` keywords?

A. 0  B. 1  C. 2  D. 3

Solution. The answer is C. Without the commented `var` keywords, all references refer to a global variable. (Gross, right?) With the keywords added, the first `x` remains global, but each function picks up its own local copy. Block scope doesn’t exist in JavaScript like it exists in Java, so the final `var` is actually superfluous. So, we go from 1 to 3 `x`’s.

Problem 2 (Object Literals) (2 points).
Consider the following buggy code, intended to keep a simple record of all of the recitation sections in 6.170. (Some lines have comments naming them for easy reference below.) Each section is associated with a TA name and a list of student names. Recall that the `push` method of arrays adds a value to the end.

```javascript
// Global variable recording the list of recitation sections.
var sections = [];

// Initialize sections with a given number of empty records.
function initSections(numSections) {
    var i, template = {ta : "Unassigned", students : []}; /* L1 */
```
Solutions to Quiz 1 (September 15)

sections = []; /* L2 */

for (i = 0; i < numSections; i += 1) {
    sections.push(template); /* L3 */
}
}

function test() {
    initSections(12);
    sections[0].ta = "Bryan"; /* L4 */
    sections[1].ta = "Vivek";
    sections[2].ta = "Rebecca";
}

After we call test(), how many different TA names are represented in ta fields within the value of sections?

A. 0  B. 1  C. 2  D. 3

Which of the following proposed changes would fix the bug? (Each suggestion is tagged with the name of one relevant code line.)

A. Instead of the push method, the loop body should use normal array subscripting. (L3)
B. Instead of assigning raw strings like "Bryan", function test() should be assigning string objects like new String("Bryan"). (L4)
C. Instead of before the loop, the assigment to template should be inside the loop. (L1)
D. Instead of creating an empty array in the initial assignment to sections, function initSections() should allocate an array of the proper size. (L2)

Solution. The answers are B and C. In initSections(), the assignment to template produces one object to be shared by all the array elements! Every assignment to a ta field overwrites all the others, in some sense.

Problem 3 (Registering Callbacks) (2 points).
Here’s a simple but buggy JavaScript implementation of a mechanism for registering callbacks, functions that should be called when some particular event occurs. The details of the functions and the event don’t matter for this example; we’re just concerned with the mechanics of saving a list of functions to be triggered later.

// Global variable storing a representation of
// all callbacks registered so far
var callbacks = function() { }; /* L1 */

// Reset the callback list.
function resetCallbacks() {
    callbacks = function() { }; /* L1 */
}

// Add a new function to the list of callbacks,
function registerCallback(f) {
    callbacks = function() { /* L2 */
        callbacks(); /* L3 */
        f();
    }
}

// Call all registered callbacks, in order.
function triggerCallbacks() {
    callbacks();
}

// A test case
function test() {
    resetCallbacks();
    registerCallback(function() { console.log("Test1"); }); /* L4 */
    registerCallback(function() { console.log("Test2"); });
    triggerCallbacks();
}

When we run the test() function, how many lines of text are logged to the console?

A. 1  B. 2  C. 3  D. something else

Which of the following proposed changes would fix the bug?

A. In function registerCallback(), begin by assigning the value of callbacks to a local variable, and use that variable instead of callbacks in the anonymous function. (L3)
B. In function registerCallback(), add var before the assignment to callbacks. (L2)
C. In function test(), replace the two anonymous functions with names, referring to new top-level functions defined to use the same bodies as the originals. (L4)
D. Change the original definition of callbacks into function callbacks() { }, so that the function is created in the first place with recursion allowed, to prepare for the recursive call that happens later. (L1)

Solution. The answers are D and A. The exact behavior will actually depend on the JavaScript implementation! There could be 0 outputs because of a stack overflow during a compilation phase before execution, or we could see many Test1 outputs.

Why? In the registerCallback() definition, the 2nd reference to callbacks is actually resolved when the callbacks are finally called. That is, in evaluating the callbacks, we begin by calling the very same function, creating an infinite loop (through function calls)! Assigning to a local variable causes us to “save a copy” of the value at the time when registerCallback() was called.