Jerry Cain
CS 106AJ

Handout #46S
November 30, 2018

Answers to Practice Final #1

Review session: **Saturday, December 8, 11:00A.M. –1:00P.M. (McCullough 115)**
Scheduled final: **Monday, December 10, 8:30–11:30A.M. (380-380C)**

Problem 1—Short answer (10 points)

1a) As written, the program leaves the array in the following state:

`list`

50	10	10	10	10

If you had wanted `mystery` to "rotate" the array elements, you would need to run the loop in the opposite order to ensure that no elements are overwritten, like this:

```
function mystery(array) {
   let tmp = array[array.length - 1];
   for (let i = array.length - 1; i > 0; i--) {
      array[i] = array[i - 1];
   }
   array[0] = tmp;
}
```

1b) Calling `conundrum` displays the value 28 on the console. The key to understanding this problem lies in figuring out which `x` and `y` values are used at each point. In the function returned by `puzzle`, the value of `x` comes from the closure and is therefore the value 17 passed to puzzle, and the value of `y` is the argument to the function `f`, which is 6. The body of the function computes 2 times `x` minus `y`, which is 28.

Problem 2—Simple graphics (15 points)

```
/**
 * Function: createPieChart
 * ------------------------
 * Creates and returns a GCompound object that represents a pie
 * chart.  Each value in the supplied data array relative to the
 * sum of all values dictates the size of each slice in the pie
 * chart.  The reference point of the entire GCompound is the pie
 * chart's center.
 */
function createPieChart(r, data) {
   let pie = GCompound();
   let total = sumArray(data);
   let start = 0;
   for (let i = 0; i < data.length; i++) {
      let fraction = data[i] / total;
      let sweep = fraction * 360;
      let slice = GArc(-r, -r, 2 * r, 2 * r, start, sweep);
      slice.setFilled(true);
      slice.setFillColor(WEDGE_COLORS[i % WEDGE_COLORS.length]);
      pie.add(slice);
      start += sweep;
   }
   return pie;
}

/**
 * Function: sumArray
 * ------------------
 * Returns the sum of all the numbers residing
 * in the supplied array. (This function would not
 * need to be written, since it appears in a
 * lecture slide and in the reader, on page 258.)
 */
function sumArray(array) {
   let sum = 0;
   for (let i = 0; i < array.length; i++) {
      sum += array[i];
   }
   return sum;
}
```

Problem 3—Interactive graphics (20 points)

```
/**
 * FifteenPuzzle.js
 * ----------------
 * This program animates the Fifteen Puzzle.
 */

/* Constants */
const SQUARE_SIZE = 60;
const INSET = 2;
const TILE_SIZE = SQUARE_SIZE - 2 * INSET;
const GWINDOW_WIDTH = 4 * SQUARE_SIZE;
const GWINDOW_HEIGHT = GWINDOW_WIDTH;
const TILE_FONT = "18px 'Times New Roman'";

/**
 * Function: FifteenPuzzle
 * -----------------------
 * Defines the factory function that manages the entire
 * simulation.
 */
function FifteenPuzzle() {
   let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
   initFifteenPuzzle(gw);
   let clickAction = function (e) {
      let tile = gw.getElementAt(e.getX(), e.getY());
      if (tile === null) return;
      if (tryToMove(gw, tile, SQUARE_SIZE, 0)) return;
      if (tryToMove(gw, tile, -SQUARE_SIZE, 0)) return;
      if (tryToMove(gw, tile, 0, SQUARE_SIZE)) return;
      tryToMove(gw, tile, 0, -SQUARE_SIZE);
   };
   gw.addEventListener("click", clickAction);
}

/**
 * Function: initFifteenPuzzle
 * ---------------------------
 * Constructs the initial state of the Fifteen Puzzle
 * board by properly implanting fifteen numbered squares
 * into the supplied GWindow.
 */
function initFifteenPuzzle(gw) {
   let x = 0;
   let y = 0;
   for (let number = 1; number < 16; number++) {
      gw.add(createNumberedSquare(number), x, y);
      if (number % 4 === 0) { // yes? time to advance down a row
         x = 0;
         y += SQUARE_SIZE;
      } else { // no? jimmy over to the right
         x += SQUARE_SIZE;
      }
   }
}
```

```
/**
 * Function: createNumberedSquare
 * ------------------------------
 * Constructs a GCompound consisting of a square (GRect) with
 * a centered number (GLabel) with a reference point set at
 * the upper left corner of the square.
 */
function createNumberedSquare(number) {
   let tile = GCompound();
   let square = GRect(0, 0, TILE_SIZE, TILE_SIZE);
   square.setFilled(true);
   square.setFillColor("LightGray");
   tile.add(square, INSET, INSET);
   let label = GLabel("" + number);
   label.setFont(TILE_FONT);
   let cx = (SQUARE_SIZE - label.getWidth())/2;
   let cy = (SQUARE_SIZE + label.getAscent())/2;
   tile.add(label, cx, cy);
   return tile;
}

/**
 * Predicate function: tryToMove
 * -----------------------------
 * Uses the upper left corner of the tile provided via the second
 * argument to compute the upper left corner of a neighboring tile.
 * If the neighboring (ulx, uly) corner is outside the bounds of the window,
 * or it's the upper left corner of an actual tile, then the original tile
 * is blocked and can't be moved, and tryToMove expresses failure by returning
 * false.  If (ulx, uly) overlays the empty square, then the original tile
 * is shifted into the void and true is returned instead.
 *
 * Because my solution is slightly more styled than that required by the
 * problem, you didn't need to add INSET + 1 to get the ulx or uly values.
 */
function tryToMove(gw, tile, dx, dy) {
   let ulx = tile.getX() + dx + INSET + 1;
   let uly = tile.getY() + dy + INSET + 1;
   if (ulx < 0 || ulx >= gw.getWidth()) return false;
   if (uly < 0 || uly >= gw.getHeight()) return false;
   if (gw.getElementAt(ulx, uly) !== null) return false; // not empty!
   tile.move(dx, dy);
   return true;
}
```

Problem 4—Strings (15 points)

```
/**
 * Predicate Function: isAnagram
 * -----------------------------
 * Returns true if and only if the two supplied
 * strings are anagrams, as per the definition of
 * anagram in the problem statement.
 */
function isAnagram(s1, s2) {
   let table1 = compileFrequencyTable(s1);
   let table2 = compileFrequencyTable(s2);
   for (let i = 0; i < 26; i++) {
      if (table1[i] !== table2[i])
         return false
   }
   return true;
}

/**
 * Function: compileFrequencyTable
 * -------------------------------
 * Compiles and returns a frequency table for the
 * supplied string.  The table is of length
 * 26, and the 0th entry ultimately contains the number
 * of a's (and A's), the 1th entry ultimately contains the
 * number of b's and B's, and so forth.
 */
function compileFrequencyTable(str) {
   let table = createArray(26, 0);
   for (let i = 0; i < str.length; i++) {
      if (isLetter(str.charAt(i))) {
         let ch = str.charAt(i).toUpperCase();
         table[ch.charCodeAt(0) - "A".charCodeAt(0)]++;
      }
   }
   return table;
}

/**
 * Predicate Function: isLetter
 * ----------------------------
 * Returns true if and only if ch is of length one, and its
 * one character is a letter of the English alphabet.  This
 * function did not need to be written out, since it
 * appears on page 239, and has appeared in several lecture slides.
 */
function isLetter(ch) {
   if (ch.length !== 1) return false;
   ch = ch.toUpperCase();
   return ch >= "A" && ch <= "Z";
}
```

```
/**
 * Function: createArray
 * ---------------------
 * Creates an array of the specified length where
 * every single element is equal to the supplied value.
 * This function did not need to be written, since it
 * appears on page 263 of the course reader, and it's also
 * appeared in one or more lecture slides.
 */
function createArray(length, value) {
   let array = [];
   for (let i = 0; i < length; i++) {
      array.push(value);
   }
   return array;
}
```

Problem 5—Arrays (10 points)

```
/**
 * Function: doubleImage
 * ---------------------
 * Accepts the fully loaded image, constructs a replica
 * of that image, save for the fact that it's twice as wide and
 * twice as tall, and then returns that image.
 */
function doubleImage(image) {
   let compact = image.getPixelArray();
   let expanded = [];
   for (let row = 0; row < compact.length; row++) {
      // introduce two rows in expanded on behalf of one in original
      expanded.push([]);
      expanded.push([]);
      for (let column = 0; column < compact[0].length; column++) {
         let pixel = compact[row][column];
         expanded[2 * row].push(pixel);
         expanded[2 * row].push(pixel);
         expanded[2 * row + 1].push(pixel);
         expanded[2 * row + 1].push(pixel);
      }
   }
   return GImage(expanded);
}
```

Problem 6—Working with data structures (15 points)

```
/**
 * Predicate function: playerSmellsWumpus
 * --------------------------------------
 * Searches the cave just enough to decide whether
 * the player is within one or two rooms of the wumpus.
 * We assume the player and wumpus are guaranteed to be
 * in distinct rooms.
 */
function playerSmellsWumpus(cave) {
   let room = cave.playerLocation;
   for (let i = 0; i < 3; i++) {
      let roomOneAway = cave.connections[room][i];
      if (roomOneAway === cave.wumpusLocation) return true;
      for (let j = 0; j < 3; j++) {
         let roomTwoAway = cave.connections[roomOneAway][j];
         if (roomTwoAway === cave.wumpusLocation) return true;
      }
   }
   return false;
}
```

Problem 7— Reading data structures from embedded XML (15 points)

```
/**
 * Function: ElectionData
 * ----------------------
 * Defines the factory function that constructs a class with two
 * exposed methods, as defined in the practice exam.  constituencyNames
 * and constituenciesMap are immediately defined where they are so that
 * they are part of the closure accessed by the implementations of
 * getConstituencyNames and getResults.
 */
function ElectionData() {
   let constituencyNames = [];
   let constituenciesMap = {};
   parseXML(constituencyNames, constituenciesMap);
   return {
      getConstituencyNames: function() { return constituencyNames; },
      getResults: function(name) {
         let results = constituenciesMap[name];
         if (results === undefined) results = [];
         return results;
      }
   };
}

/**
 * Function: parseXML
 * ------------------
 * Accepts the empty constituencyNames array and the
 * empty constituenciesMap map, and populates them
 * with the names of all constituencies and the collection
 * of key/value pairs that represent the relevant
 * constituent-name -> election-results information.
 */
function parseXML(constituencyNames, constituenciesMap) {
   let electionXML = document.getElementById("ElectionData");
   let constituencies = electionXML.getElementsByTagName("constituency");
   for (let i = 0; i < constituencies.length; i++) {
      let constituency = constituencies[i];
      let constituencyName = constituency.getAttribute("name");
      constituencyNames.push(constituencyName);
      constituenciesMap[constituencyName] = [];
      let candidates = constituency.getElementsByTagName("candidate");
      for (let j = 0; j < candidates.length; j++) {
         let candidate = candidates[j];
         let entry = {
            candidate: candidate.getAttribute("name"),
            party: candidate.getAttribute("party"),
            votes: parseInt(candidate.getAttribute("votes"))
         };
         constituenciesMap[constituencyName].push(entry);
      }
   }
}
```