

Mechanics of Functions (Addendum)

Mechanics of Functions (Addendum)

Jerry Cain and Eric Roberts
CS 106J
April 19, 2017

Exercise: Generating Prime Factorizations

- A more computationally intense problem is to generate the prime factorization of a positive integer n .
- An integer is prime if it's greater than 1 and has no positive integer divisors other than 1 and itself.
 - ✓ 5 is prime: it's divisible only by 1 and 5.
 - ✓ 6 is not prime: it's divisible by 1, 2, 3, and itself.
- Some prime factorizations:

```

-> PrimeFactorizations(501, 512)
501 = 3 * 167
502 = 2 * 251
503 = 503
504 = 2 * 2 * 2 * 3 * 3 * 7
505 = 5 * 101
506 = 2 * 11 * 23
507 = 3 * 13 * 13
508 = 2 * 2 * 127
509 = 509
510 = 2 * 3 * 5 * 17
511 = 7 * 73
512 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2
->
    
```

PrimeFactorizations.js

```

function constructFactorization(n) {
  var result = n + " = ";
  var first = true;
  var factor = 2;

  while (n > 1) {
    if (isDivisibleBy(n, factor)) {
      if (!first) result += " * ";
      first = false;
      result += factor;
      n /= factor;
    } else {
      factor++;
    }
  }

  return result;
}
    
```

PrimeFactorizations.js

Some thought questions and exercises:

- The solution relies on a single Boolean called **first**. What problem is **first** solving for us?
- During our trace of **constructFactorization(180)**, **factor** assumed the values of 2, 3, 4, and 5. 2, 3, and 5 are prime numbers and therefore qualified to appear in a factorization? How does the implementation guarantee 4 will never make an appearance in the returned factorization?
- What is returned by **constructFactorization(1)**? How could you have changed the implementation to return **"1 = 1"** as a special case return value?
- Trace through the execution of **constructFactorization(363)** as we did for **constructFactorization(180)**.
- Our implementation relies on a parameter named **n** to accept a value from the caller, and then proceeds to destroy **n** by repeatedly dividing it down to 1. Does this destruction of **n** confuse **PrimeFactorizations's** for loop? Note that its counting variable is also named **n**.

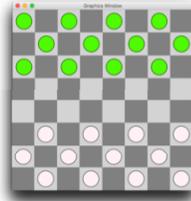
Exercise: Drawing A Checkerboard

- For the rest of lecture, we'll collectively design and decompose (and to the extent we have time, implement) a graphics program that draws the initial configuration for a game of checkers.

```

function DrawCheckerboard() {
  // ...
  // Defines the entry point to the entire program, and subdivides
  // the entire problem into three parts: creating and presenting
  // a properly sized window, drawing a standard checkerboard within it,
  // and then layering 24 checkers on top of that board.
}

function DrawCheckerboard() {
  var gw = GWindow(BOARD_WIDTH, BOARD_HEIGHT);
  drawBoard(gw);
  drawCheckers(gw);
}
    
```



DrawCheckerboard.js

```

function drawBoard(gw) {
  // ...
  // Draws a 10x10 checkerboard.
}
    
```

DrawCheckerboard.js

```
function drawCheckerboard() {
  // Function: drawCheckerboard
  // .....
  // Define the entry point to the entire program, and subdivide
  // the entire program into three parts: creating and processing
  // a properly sized window, drawing a standard checkerboard within it,
  // and then layering 24 checkers on top of that board.
  // .....
  var sp = new window(BOARD_WIDTH, BOARD_HEIGHT);
  drawBoard(sp);
  drawSquares(sp);
}

function drawBoard() {
  // Function: drawBoard
  // .....
  // Draw the standard BOARD_DIMENSION by BOARD_DIMENSION checkerboard.
  // Note that the board's origin--we'll call it (0, 0)--is the upper left corner.
}

function drawSquares(sp) {
  // Function: drawSquares
  // .....
  for (var row = 0; row < BOARD_DIMENSION; row++) {
    for (var col = 0; col < BOARD_DIMENSION; col++) {
      drawSquare(sp, row, col, getSquareColor(row, col));
    }
  }
}
```

DrawCheckerboard.js

```
function drawSquare(sp, row, col, color) {
  // Function: drawSquare
  // .....
  // Draw a single checkerboard square at the specified board coordinate.
  // .....
  var size = col * SQUARE_WIDTH;
  var top = row * SQUARE_HEIGHT;
  square.setAttribute(
    "width", size);
    square.setAttribute(
    "height", size);
    sp.appendChild(
    square);
}

function getSquareColor() {
  // Function: getSquareColor
  // .....
  // Because one of the two colors used to draw checkerboard squares,
  // because one of the two colors used to draw checkerboard squares,
  // because we want the standard checkerboard pattern, we explicit
  // the modulus characteristic of row + col to decide which of the
  // color constants to return.
}

function getSquareColor(row, col) {
  return (row + col) % 2 == 0 ? LIGHT_SQUARE_COLOR : DARK_SQUARE_COLOR;
}
```

DrawCheckerboard.js

```
function drawSquares(sp) {
  // Function: drawSquares
  // .....
  // Place the first player's checkers in the upper three rows of the board,
  // and then place the second player's checkers in the lower three rows of the board.
  // .....
  drawRowOfCheckers(sp, 0, 2, PLAYER_ONE_COLOR);
  drawRowOfCheckers(sp, BOARD_DIMENSION - 3, BOARD_DIMENSION - 1, PLAYER_TWO_COLOR);
}

function drawRowOfCheckers(sp, start, stop, color) {
  // Function: drawRowOfCheckers
  // .....
  // Draw row of checkers in alternating columns in the row numbered
  // start up through and including stop. The checkers themselves are
  // colored in each square and filled with the specified color.
  // .....
  for (var row = start; row <= stop; row++) {
    for (var col = 0; col < BOARD_DIMENSION; col++) {
      if (shouldDrawChecker(row, col)) {
        drawChecker(sp, row, col, color);
      }
    }
  }
}
```

DrawCheckerboard.js

```
function shouldDrawChecker(row, col) {
  // Predicate Function: shouldDrawChecker
  // .....
  // Returns true if and only if a checker should be placed at the
  // specified row, col coordinate.
}

function shouldDrawChecker(row, col) {
  return (row + col) % 2 == 0;
}

function drawChecker() {
  // Function: drawChecker
  // .....
  // Draw a single checker at the provided (row, col) coordinate.
  // The outline color of the checker is always black, but the fill
  // color is dictated by the final parameter.
}

function drawChecker(sp, row, col, color) {
  var size = col * 0.5 * SQUARE_WIDTH;
  var cy = (row + 0.5) * SQUARE_HEIGHT;
  drawCircleAndCircle(sp, cy, cy, CHECKER_RADIUS, color);
}
```

DrawCheckerboard.js

```
function drawCircleAndCircle(sp, cx, cy, radius, fillColor) {
  // Function: drawCircleAndCircle
  // .....
  // Place a circle at the specified radius so that its center
  // overlaps the pixel-based coordinate (cx, cy). The circle's
  // border color is always black, but the fill color is dictated
  // by the value supplied through the final parameter.
  // .....
  var circle = new Canvas(
    radius, radius, radius, radius);
  circle.setAttribute(
    "fill", fillColor);
  circle.setAttribute(
    "stroke", "black");
  circle.setAttribute(
    "stroke-width", 1);
  sp.appendChild(
  circle);
}
```