1 First-Class Functions in Depth

1. Introduction: a discussion of what you can and cannot do with first-class functions. Sometimes, not being able to do something is a feature.

2. Recap some higher-order functions (writing their types):
   (a) app (consumes a function),
   (b) A function that returns another function:

   ```javascript
   function double(x) { return x; }  
   function produceDouble(x) { return double; }
   ```

   (c) You can nest functions within each other (put `double` inside `produceDouble`)
   (d) Useful combination of nesting and returning functions (`makeAdder`). Show that each call does not “overwrite the outer value”.
   (e) You don’t have to name functions (refactor `makeAdder`). Go over using lambdas with map and filter.
   (f) Explain closures and go over their notation using the `add` function in `makeAdder`. I want to write

   ```javascript
   makeAdder(10) === add [ x -> 10 ]
   ```

   (g) Predict the output of the `crazy` function below:

   ```javascript
   function crazy(x) {
     function add(y) {
       let tmp = x + 1;
       x = x + 1;
       return tmp + y;
     }
     return add;
   }
   let f = crazy(100);
   let g = crazy(100);
   console.log(f(100));
   console.log(f(100));
   console.log(g(100));
   ```

3. What do functions actually do? Why do we write functions?
   (a) They let you parameterize a block of code. (Consider writing several add functions.)
   (b) They delay evaluation. Code inside a function does not execute until it is applied.

   ```javascript
   function F(f) {
     return 10;
   }
   F(function() {
     console.log("hi");
   });
   ```

4. Closures can be used to hide information:
// guessingGame(secret: number) => (guess: number) => undefined
function guessingGame(secret) {
    function game(guess) {
        if (guess === secret) {
            console.log("WINNER");
        } else if (guess < secret) {
            console.log("Guess higher");
            return game;
        } else {
            console.log("Guess lower");
            return game;
        }
    }
    return game;
}

let guesser = guessingGame(4383); // game[ secret -> 4383 ]
guesser(100) // === game[ secret -> 4383 ]

Play the guessing game by using X % Y as the secret.

5. The idea of curried functions: write original map and then write the addTenAll function. Show the curried variant of map and show that addTenAll becomes much easier to write.

function map(f) {
    return function(a) {
        let r = [];
        for (let i = 0; i < a.length; ++i) {
            r.push(f(a[i]));
        }
        return r;
    }
}

let addTenAll = map(function(n) { return n + 10; });

Functions don’t need to take more than one argument.

6. Example problem: given an input that does not use console.log:

function foo(f) {
    f(function(x) {
        console.log("a");
    });
}