Midterm 1 In-Class Review Problems

Problem 1: Tracing Execution with Higher-Order Functions

What is the output of the following function?

```javascript
function app(f, g) {
  return f(function() { console.log("hat"); g("mat") });
}

app(function(h) {
  h();
  console.log("cat");
},
  function(x) { console.log(x); });
```

**Approach** First, let’s rewrite the program, giving names to all the anonymous functions. This will make it easier to reason about the program.

```javascript
function app(f, g) {
  function A() { console.log("hat"); g("mat") });
  return f(A);
}

function B(h) {
  h();
  console.log("cat");
}

function C(x) {
  console.log(x);
};

app(B, C);
```

Check that the two programs above are equivalent. Now, we can reason as follows.

1. The program defines three functions and then calls `app(B, C).

2. Within `app`, the value of the arguments are `f → B` and `g → C`. The value of the local function `A` is the closure `A[f → B, g → C]`. Since body of `A` does not use `f`, we can safely write this closure-value as `A[g → C].`

3. Within `app`, we call `f(A)`. Based on the reasoning above, this is really the call `B(A[g → C])`, so we enter `B` with `A[g → C]` as the value of `h`.

4. Within `B`, there are two statements:

   (a) we call `h()`, which is really the call `A[g → C]()`, so we enter `A[g → C].`
i. Within \( A[g \rightarrow C] \) the program **first displays** “hat” and then calls \( g("mat") \), which is really the call \( C("mat") \); thus we enter \( C \) with “mat” as the value of \( x \).

ii. Within \( C \), we display the value of the argument \( x \), thus the program displays “mat” next.

\[ f(A) \]. From above,

(b) \( B \) then displays “cat”.

**Problem 2: Functional Inputs**

Provide inputs to the function below that make the program display “A” and then “B”. Your answer may not use `console.log`

```javascript
function foo(f, g) {
  function H(x) {
    if (x > 10) { console.log("A"); }
  }

  function J(y) {
    function I() { console.log(y); }
    g(I);
  }
  f(H, J);
}
```

**Approach** The first step in this kind of problem is to figure out the types of all functions. It is obvious that \( x \) is a number, \( y \) is a string, \( f \) is a function of two arguments, and \( g \) is a one-argument function. So, we can write what we have so far as follows:

```javascript
// foo(f: (k1: ____, k2: ____)) => undefined,
// g: (j: ____)) => undefined): undefined
function foo(f, g) {
  // H(x: number): undefined
  function H(x) {
    if (x > 10) { console.log("A"); }
  }

  // J(y: string): undefined
  function J(y) {
    // I()): undefined
    function I() { console.log(y); }
    g(I);
  }
  f(H, J);
}
```

Notice that \( f \) is applied to \( H \) and \( J \), so the types of its arguments must be the same as the types of \( H \) and \( J \) respectively. Similarly, since \( g \) is applied to \( I \), the type of the argument of \( g \) must be the same as the type of \( I \). This gives us complete type information:

```javascript
// foo(f: (k1: (x: number) => undefined, k2: (y: string) => undefined) => undefined,
// g: (j: () => undefined) => undefined): undefined
function foo(f, g) {
  // H(x: number): undefined
  function H(x) {

  }
```


function H(x) {
    if (x > 10) { console.log("A"); }
}

// J(y: string): undefined
function J(y) {
    // I(): undefined
    function I() { console.log(y); }
    g(I);
}

f(H, J);

Now, we can write a sketch of our solution based entirely on the types:

function f(k1, k2) {
    k1(11);
    k2("B");
}

function g(j) {
    j();
}

foo(f, g)

Since the types are correct, the program above will run without errors, but will not display anything. However, we can now start to reason through the execution of the program.

1. foo(f, g) calls f(H, J), so we enter f with k1 -> H and k2 -> J.

2. The body of H prints “A” if x > 10, so we should change f to call k1(11), which is really the call H(11)

3. The function J takes a string y as an argument, and we can call it from f too, but it is not clear what the string should be. Let’s guess "B", so f now also calls k2("B"), which is really J("B").

4. Within J is the call g(I). However, the value of I is I[y -> "B"]. Therefore, we enter g, with j -> I[y -> "B"].

5. Therefore, we should modify g to call its argument j(), which is really the call I[y -> "B"](), which prints “B”.

The final solution is:

function f(k1, k2) {
    k1(11);
    k2("B");
}

function g(j) {
    j();
}

foo(f, g)

You should check that this program displays what is expected by reasoning through its execution as we did in Problem 1.
Problem 3: Object References

What does this program display?

```javascript
let o = { x: 10 }; // let arr = [ o ];
arr.push(o);
arr.push(o);
arr[2].x = 20;
o = { x: 30 };
console.log(arr[0].x + arr[1].x + arr[2].x);
```

Approach

1. The program creates an array that initially has a single element, which is a copy of the object-reference stored in `o`.
2. The program then pushes two copies of the same object-reference into the array. Note that at this point, there is exactly one object, but four references (count them) to that object (three references in `arr`, and one in the variable `o`).
3. The program then updates the field `x` of that object to 20. (It uses the reference `arr[2]` to do so, but that is not relevant.)
4. Next, the program creates a new object `{ x: 30 }` and updates the variable `o` to store a reference to this new object. However, this update does not affect the contents of the array, which are references to the first object.
5. Therefore, the program displays 60.

1 Problem 4: More Object References

This alternate example is meant to demonstrate the difference between using an object reference and using a new object. What does the following program display?

```javascript
let o = { x: 10 }; // let arr = [ o ];
arr.push( { x: 10 } );
arr.push( { x: 10 } );
arr[2].x = 20;
o = { x: 30 };
console.log(arr[0].x + arr[1].x + arr[2].x);
```

Approach

1. Line 1: The program creates the object in memory and stores a reference to that object in variable `o`.
2. Line 2: The program creates a 1-element array that initially holds a copy of the value stored in `o` (which is a reference to the object in memory).
3. Lines 3 and 4: Each line creates a new object in memory and stores a reference to that object in the array. Note that at this point, there are three objects, but two references to the first object, one reference to the second object, and one reference to the third object.
4. Line 5: The program updates the field x of one object to 20. (It uses the reference arr[2] to do so, *this is relevant.*

5. Line 6: The program creates a new object `{ x: 30 }` and updates the variable o to store a reference to this new object. *However, this update does not affect the contents of the array, which contains a reference to the first object.* At this point, there are four objects in memory and one reference to each object.