COMP 333
Summer 2025
Practice Exam #2 (Solutions)

This is representative of the kinds of topics and kind of questions you may be asked on the midterm. This practice exam, along with the in-class handouts covering JavaScript, as well as the guide to object-oriented programming in JavaScript, are intended to be comprehensive of everything on the exam. That is, I will not ask anything that's not somehow covered by those sources.

You are permitted to bring two 8.5 x 11 sheets of paper into the exam with you, as long as they have handwritten notes on them. Both sides of both sheets can be used. To be clear, these must be entirely handwritten.

Higher-Order Functions in JavaScript

1.) Write the output of the following JavaScript code:

```
function foo(fooParam) {
  return function (innerParam) {
    return fooParam - innerParam;
  }
}

let f1 = foo(7);      // fooParam = 7 for f1
let f2 = foo(10);     // fooParam = 10 for f2
console.log(f1(2));   // innerParam = 2 for f1; 7 - 2 = 5
console.log(f2(3));   // innerParam = 3 for f2; 10 - 3 = 7
console.log(f1(4));   // innerParam = 4 for f1; 7 - 4 = 3
console.log(f2(5));   // innerParam = 5 for f2; 10 - 5 = 5
```

```
5
7
3
5
```

2.) Write the output of the following JavaScript code:

```
// Representing lists as higher-order functions.
// The function returns true if the given element exists in
// the list, else false (e.g., contains).  The list _is_
// the function.

function nil() {
    // nil doesn't contain any elements, so it definitely
    // doesn't contain searchKey, either
    return function (searchKey) {
        return false;
    };
}

function cons(head, tail) {
    // cons contains the given element searchKey, if either the
    // head of the list is searchKey, or if the tail of the list
    // contains searchKey
    return function (searchKey) {
        if (searchKey === head) {
            return true;
        } else {
            return tail(searchKey);
        }
    }
}

let emptyList = nil();
let one = cons(1, nil());
let oneTwo = cons(1, cons(2, nil()));

console.log(emptyList(1)); // false
console.log(one(1));       // true
console.log(oneTwo(1));    // true

console.log();             // <newline>

console.log(emptyList(2)); // false
console.log(one(2));       // false
console.log(oneTwo(2));    // true
```

3.) Consider the following JavaScript code with corresponding output, which calls an unseen function called `mystery`:

```
function output() {
    console.log("foo");
}

let f1 = mystery(output);
f1();
console.log();

let f2 = mystery(f1);
f2();
console.log();

let f3 = mystery(f2);
f3();
console.log();
```

Output:

```
foo
foo

foo
foo
foo
foo

foo
foo
foo
foo
foo
foo
foo
foo
```

Define the `mystery` function.

```
function mystery(f) {
    return function() {
        f();
        f();
    };
}
```

4.) Write the output of the following JavaScript code:

```
// returns a function that will bound the output of the wrapped
// function, so the output is never less than min or greater
// than max
function cap(min, max, wrapped) {
  return function (param) {
    let temp = wrapped(param);
    if (temp < min) {
      return min;
    } else if (temp > max) {
      return max;
    } else {
      return temp;
    }
  };
}

function addTen(param) {
  return param + 10;
}

function subTen(param) {
  return param - 10;
}

let f1 = cap(0, 10, addTen);
let f2 = cap(0, 100, addTen);
let f3 = cap(0, 10, subTen);
let f4 = cap(0, 100, subTen);

console.log(f1(0)); // output: 10
console.log(f1(5)); // output: 10
console.log(); // prints an empty line

console.log(f2(0)); // output: 10
console.log(f2(5)); // output: 15
console.log(); // prints an empty line

console.log(f3(0)); // output: 0
console.log(f3(5)); // output: 0
console.log(); // prints an empty line

console.log(f4(0)); // output: 0
console.log(f4(5)); // output: 0
console.log(); // prints an empty line
```

5.) Consider the following incomplete JavaScript code and output, which calls an unprovided function named `invert`:

```
function greaterThanFive(input) {
  return input > 5;
}

let notGreaterThanFive = invert(greaterThanFive);
let notEqualsFoo = invert(function (e) { return e === "foo"; });

console.log(notGreaterThanFive(3));
console.log(notGreaterThanFive(7));
console.log(notEqualsFoo("foo"));
console.log(notEqualsFoo("bar"));


---OUTPUT---
true
false
false
true
```

`invert` should return a new function that effectively inverts the behavior of the provided function. `invert` should work for any function taking one parameter and returning a boolean, not just the calls shown here. Implement `invert` below.

```
function invert(f) {
  return function (param) {
    return !f(param);
  }
}
```

6.) Consider the following JavaScript code and output:

```
function printAndReturn(e) {
    console.log("Value: " + e);
    return e;
}

ifNotNull(1, printAndReturn);
console.log(ifNotNull(1, printAndReturn));

ifNotNull(null, printAndReturn);
console.log(ifNotNull(null, printAndReturn));

console.log(
    ifNotNull(1 + 1,
              a => ifNotNull(2 + 2,
                             b => a + b)));
console.log(
    ifNotNull(7,
              function (e) {
                  console.log(e);
                  return ifNotNull(null,
                                   function (f) {
                                       console.log(f);
                                       return 8;
                                   })
              }));
```

Output:

```
Value: 1
Value: 1
1
null
6
7
null
```

`ifNotNull` takes two parameters:

1. Some arbitrary value, which might be `null`
2. A function. This function is called with the arbitrary value if the value is not `null`, and the result of the function is returned. If the value is `null`, this function isn't called, and `null` is returned instead.

Define the `ifNotNull` function on the next page, so that the output above is produced.

```
function ifNotNull(value, f) {
    if (value !== null) {
        return f(value);
    } else {
        return value;
    }
}
```

7.) Consider the following JavaScript code and output:

```
function add(x, y) {
  return x + y;
}

function subtract(x, y) {
  return x - y;
}

let debugAdd = debug(add);
let debugSubtract = debug(subtract);

let addResult = debugAdd(1, 2);
let subtractResult = debugSubtract(5, 1);

console.log(addResult);
console.log(subtractResult);
```

Output:

```
First param: 1
Second param: 2
Return value: 3
First param: 5
Second param: 1
Return value: 4
3
4
```

The `debug` function takes another function taking two parameters, and returns a new function taking two parameters. The returned function prints its parameters, calls the wrapped function, prints the wrapped function's return value, then returns the wrapped function's return value. Implement `debug` below.

```
function debug(f) {
    return function (x, y) {
        console.log("First param: " + x);
        console.log("Second param: " + y);
        let retval = f(x, y);
        console.log("Return value: " + retval);
        return retval;
    }
}
```

8.) Consider the following JavaScript code:

```
function printHello() {
    console.log("hello");
}

function dontCall() {
    return function (f) {}
}

function callOncePlusN(n) {
    return function (f) {
        f();
        n(f);
    }
}

let zeroCalls = dontCall();
let oneCall = callOncePlusN(zeroCalls);
let twoCalls = callOncePlusN(oneCall);
let threeCalls = callOncePlusN(twoCalls);

zeroCalls(printHello);
oneCall(printHello);
twoCalls(printHello);
threeCalls(printHello);
```

Write the output of this code below:

```
hello
hello
hello
hello
hello
hello
```

9.) Consider the following JavaScript code:

```
function extractMin(arr1, arr2) {
    for (let index = 0; index < arr1.length; index++) {
        if (arr1[index] < arr2[index]) {
            arr2[index] = arr1[index];
        }
    }
}

function extractMax(arr1, arr2) {
    for (let index = 0; index < arr1.length; index++) {
        if (arr1[index] > arr2[index]) {
            arr2[index] = arr1[index];
        }
    }
}
```

`extractMin` and `extractMax` are intended to take two arrays of the same length. For each index `i` in both arrays, either the smaller or the larger of `arr1[i]` and `arr2[i]` will assigned into `arr2[i]`.

A significant amount of code is duplicated between these two functions. Write a new function named `extract` that generalizes over `extractMin` and `extractMax`, where `extract` takes:

- The first array
- The second array
- A function which takes two numbers and returns a boolean

After implementing `extract`, rewrite `extractMin` and `extractMax` to call `extract`.

```
function extract(arr1, arr2, f) {
    for (let index = 0; index < arr1.length; index++) {
        if (f(arr1[index], arr2[index])) {
            arr2[index] = arr1[index];
        }
    }
}
function extractMin(arr1, arr2) {
    extract(arr1, arr2, (a, b) => a < b);
}
function extractMax(arr1, arr2) {
    extract(arr1, arr2, (a, b) => a > b);
}
```

10.) Consider the following array definition in JavaScript:

```
let arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

10.a) Use `filter` to get an array of all even elements in `arr`.

```
// filter takes a function that takes an element and returns
// true if the element should be in the returned array, else
// false
arr.filter(e => e % 2 === 0)

// alternative answer
arr.filter(function (element) {
  return element % 2 === 0;
})
```

10.b) Use `map` to get an array of strings, where each string represents a number in `arr`. As a hint, you can call the `toString()` method on a number (e.g., `5.toString()`) in JavaScript to get its string representation.

```
// map takes a function that takes an element and returns the
// corresponding value which should be in the output array
arr.map(e => e.toString())

// alternative answer
arr.map(function (element) {
  return element.toString()
});
```

10.c) Use `reduce` to get the last element in `arr`.

```
// reduce takes a function that takes an accumulator and an
// element, and returns the value of the new accumulator.  In
// this case, reduce is only given the function, so it will use
// the first array element as the initial accumulator, and start
// iterating from the second array element
arr.reduce((accum, element) => element)

// alternative anser
arr.reduce(function (accum, element) {
  return element;
})
```

10.d) Use a combination of `filter` and `reduce` to get the sum of all elements in `arr` which are greater than `5`.

```
// this use of reduce uses an explicit starting accumulator of 0
arr.filter(e => e > 5)
   .reduce((accum, element) => accum + element, 0)

// alternative answer
arr.filter(function (e) { return e > 5 })
   .reduce(function (a, e) { return a + e }, 0)
```

Objects in JavaScript

11.) Create an object holding a field named "`foo`", holding the value `12`.

```
{ 'foo': 12 }
```

12.) Consider the following JavaScript code:

```
let obj1 = { 'foo': 1 };
let obj2 = { 'foo': true, 'bar': 3.14 };
console.log(obj1.foo);
console.log(obj1.bar);
console.log(obj2.foo);
console.log(obj2.bar);

let obj3 = { 'baz': 'hello', '__proto__': obj2 };
console.log(obj3.foo);
console.log(obj3.bar);
console.log(obj3.baz);

let obj4 = { 'foo': 3, '__proto__': obj1 };
console.log(obj4.foo);
console.log(obj4.bar);
console.log(obj4.baz);
```

What is the output of this code?

```
1
undefined
true
3.14
true
3.14
hello
3
undefined
undefined
```

13.a.) Define a constructor for `Dog` objects, where each `Dog` object has a name. An example code snippet is below, illustrating usage:

```
let d = new Dog("Rover"); // line 1
console.log(d.name);      // line 2; prints Rover
```

```
// From line 1, we need a Dog constructor that takes one
// parameter. From line 2, the constructor must be setting the
// name field of Dog objects to the parameter.
function Dog(param) {
  this.name = param;
}
```

13.b.) Define a different constructor for `Dog`, which puts a bark method **directly** on the `Dog` objects. The `bark` method should print `"Woof!"` when called. Example usage is below:

```
let d = new Dog("Sparky");
d.bark(); // prints Woof!
```

```
function Dog(name) {
  this.name = name; // not explicitly required based on the
                    // question
  // bark is directly on created Dog objects, as opposed to
  // being on the prototype chain for Dog objects
  this.bark = function() { console.log("Woof!"); }
}
```

13.c.) Define a method named `growl` for `Dog` objects, which prints `"[dog name] growls"` when called. Use `Dog`'s **prototype**, instead of putting the method directly on `Dog` objects themselves. Example usage is below:

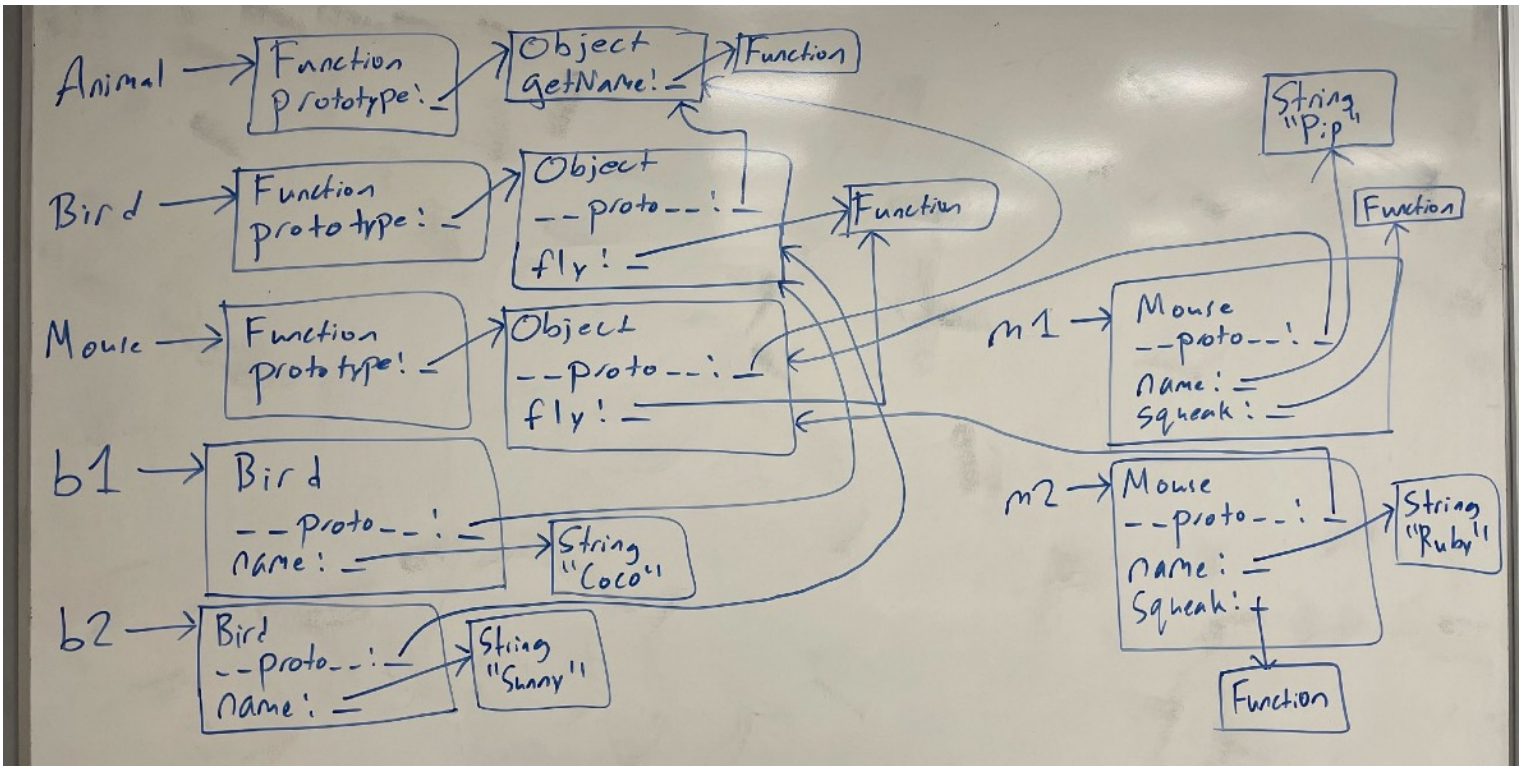
```
let d = new Dog("Rocky");
d.growl(); // prints Rocky growls
```

```
Dog.prototype.growl = function() {
  // assumes constructor initializes this.name, as with 1.a
  console.log(this.name + " growls");
}
```

14.) Consider the JavaScript code below:

```
function Animal(name) { this.name = name; }
Animal.prototype.getName = function() { return this.name; }
function Bird(name) { this.name = name; }
Bird.prototype = { '__proto__': Animal.prototype };
Bird.prototype.fly = function() {
  console.log(this.getName() + " flies");
}
function Mouse(name) {
  this.name = name;
  this.squeak = function() {
    console.log(this.name + " squeaks");
  }
}
Mouse.prototype = { '__proto__': Animal.prototype };
Mouse.prototype.fly = Bird.prototype.fly;
let b1 = new Bird("Coco"); let b2 = new Bird("Sunny");
let m1 = new Mouse("Pip"); let m2 = new Mouse("Ruby");
```

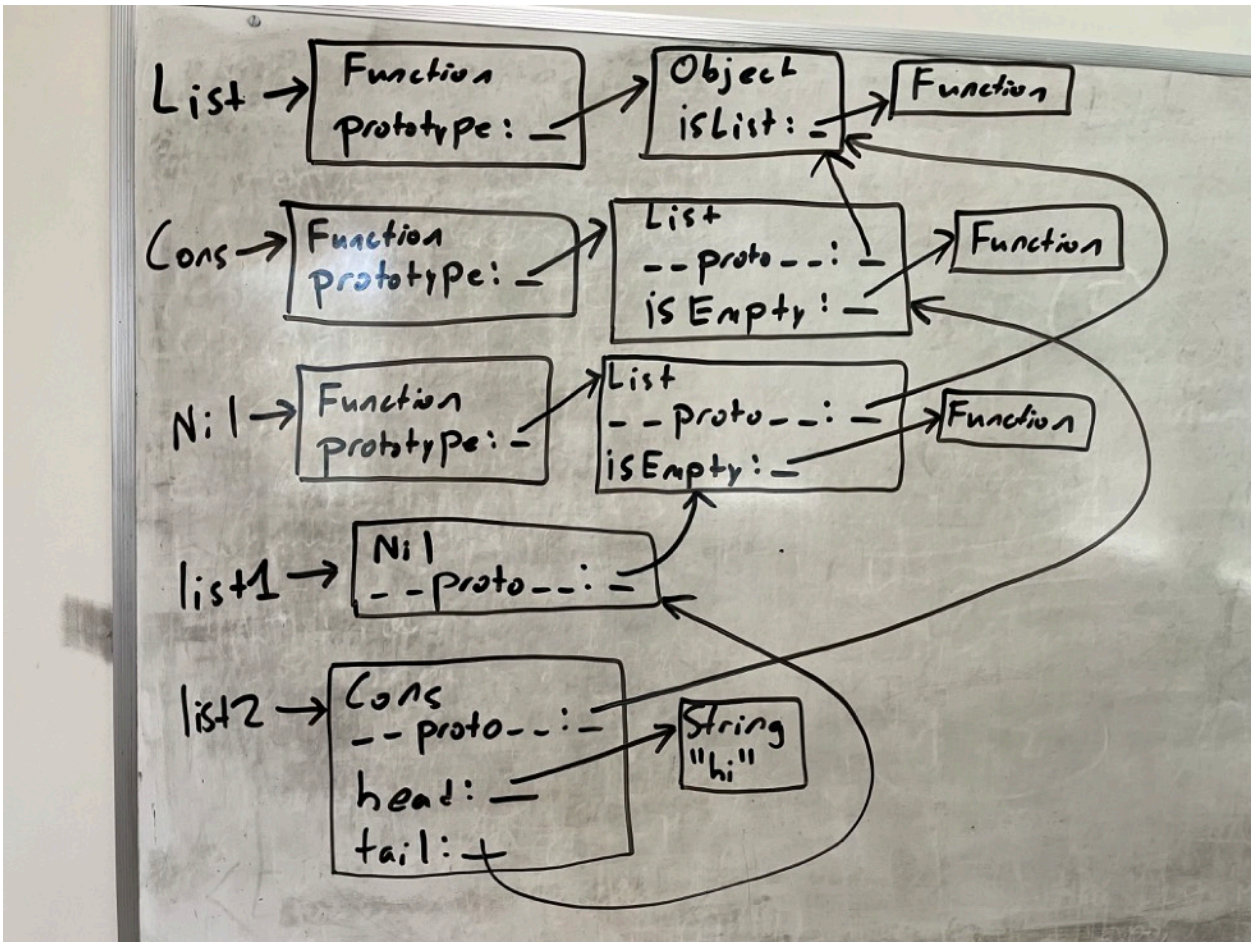
Write a memory diagram which shows how memory looks after this program executes. Your diagram should include the objects and fields associated with `b1`, `b2`, `m1`, `m2`, `Mouse`, `Bird`, and `Animal`.



15.) Consider the JavaScript code below, which implements immutable linked lists:

```
function List() {}
List.prototype.isList = function() { return true; }
function Cons(head, tail) {
  this.head = head;
  this.tail = tail;
}
Cons.prototype = new List();
Cons.prototype.isEmpty = function() { return false; }
function Nil() {}
Nil.prototype = new List();
Nil.prototype.isEmpty = function() { return true; }
let list1 = new Nil();
let list2 = new Cons("hi", list1);
```

Write a memory diagram which shows how memory looks after this program executes. Your diagram should include the objects and fields associated with `List`, `Cons`, `Nil`, `list1`, and `list2`.



16.) Consider the JavaScript code and corresponding output below:

```
let obj1 = new Obj("foo");
console.log(obj1.field); // output: foo

let obj2 = new Obj("bar");
console.log(obj2.field);          // output: bar
console.log(obj2.doubleField()); // output: barbar

let obj3 = new Obj("baz");

console.log(obj3.field); // prints baz

// hasOwnProperty is a built-in method which returns true if the
// object has the field directly, or false if it merely inherits
// the field.
console.log(obj3.hasOwnProperty("doubleField")); // prints false
```

Complete any missing elements needed to allow this code to run and produce this output.

```
// Object is a built-in in JavaScript, but not Obj. This
// requires a custom constructor. From obj1, we know that Obj
// must be a constructor, and that Obj objects need a field
// named "field".  The value of this field must be equal to
// whatever its parameter is.
function Obj(param) {
  this.field = param;
}

// From obj2, we know that we need a doubleField method on Obj
// objects.  From obj3, we know that doubleField cannot be
// directly on the Obj objects, so we must put it on Obj's
// prototype.
Obj.prototype.doubleField = function() {
  // + in this context performs string concatenation; this
  // concatenates this.field onto itself
  return this.field + this.field;
}
```

17.) Consider the JavaScript code below and corresponding output:

```
let three = new MyNumber(3);
let five = new MyNumber(5);

let eight = three.add(five);
let fifteen = three.multiply(five);

console.log(three.getValue());
console.log(five.getValue());
console.log(eight.getValue());
console.log(fifteen.getValue());


---OUTPUT---
3
5
8
15
```

Implement any missing code necessary to produce the above output. The next page is blank in case you need it.

Looking at the above code:

- There must be a `MyNumber` constructor which takes a parameter
- There must be an `add` method defined on `MyNumber` objects, either directly on the object, or on the prototype.
    - `add` takes another `MyNumber` object and returns something
- There must be a `multiply` method defined on `MyNumber` objects, either directly on the object, or on the prototype.
    - `multiply` takes another `MyNumber` object and returns something
- There must be a `getValue` method defined on `MyNumber` objects, which appears to return the number passed in the constructor
    - It looks like `add` and `multiply` are specifically returning `MyNumber` objects which wrap around the results of the operations

From there, we can take these notes and derive the code on the next page.

```
// There must be a MyNumber constructor which takes a parameter
function MyNumber(value) {
    this.value = value;
}

// There must be an add method defined on MyNumber objects,
// either directly on the object, or on the prototype.
// add takes another MyNumber object and returns something
// It looks like add and multiply are specifically returning
// MyNumber objects which wrap around the results of the
// operations
MyNumber.prototype.add = function (other) {
    return new MyNumber(this.value + other.value);
};

// There must be a multiply method defined on MyNumber objects,
// either directly on the object, or on the prototype.
// multiply takes another MyNumber object and returns something
// It looks like add and multiply are specifically returning
// MyNumber objects which wrap around the results of the
// operations
MyNumber.prototype.multiply = function (other) {
    return new MyNumber(this.value * other.value);
};

// There must be a getValue method defined on MyNumber objects,
// which appears to return the number passed in the constructor
MyNumber.prototype.getValue = function () {
    return this.value;
}
```