COMP 333
Summer 2025
Practice Exam #2

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);
let f2 = foo(10);
console.log(f1(2));
console.log(f2(3));
console.log(f1(4));
console.log(f2(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));
console.log(one(1));
console.log(oneTwo(1));

console.log();

console.log(emptyList(2));
console.log(one(2));
console.log(oneTwo(2));
```

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.

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

```
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));
console.log(f1(5));
console.log(); // prints an empty line

console.log(f2(0));
console.log(f2(5));
console.log(); // prints an empty line

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

console.log(f4(0));
console.log(f4(5));
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.

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.

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.

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:

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`.

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`.

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.

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

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

Objects in JavaScript

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

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
```

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!
```

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
```

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.

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.