

COMP 333 Practice Exam

This is representative of the kinds of topics and kind of questions you may be asked on the midterm.

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.) Consider the following JavaScript code:

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

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

function empty() {}

let f1 = rec(rec(base()));
let f2 = rec(rec(rec(base())));
f1(empty);
f2(empty);
```

How many times is `empty` called in total in the above code?

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 below.

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();
```

```
console.log(f2(0));  
console.log(f2(5));  
console.log();
```

```
console.log(f3(0));  
console.log(f3(5));  
console.log();
```

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

5.) Consider the following JavaScript code and output:

```
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:

```
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 below, so that the output above is produced.

6.) Consider the following array definition in JavaScript:

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

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

6.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.

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

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

Prototype-Based Inheritance in JavaScript

7.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
```

7.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!
```

7.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
```

8.) Consider the JavaScript code below:

```
function Animal(name) { this.name = name; }
Animal.prototype.getName = function() { return this.name; }
function Bird(name) { Animal.call(this, name); }
Bird.prototype = Object.create(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 = Object.create(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.prototype`, and `Bird.prototype`, `Animal.prototype`. As a hint, the `__proto__` field on objects refers to the corresponding object's prototype.

9.) Consider the test suite below, using `assertEquals` from the second assignment:

```
function test1() {
  let t1 = new Obj("foo");
  assertEquals("foo", t1.field);
}

function test2() {
  let t2 = new Obj("bar");
  assertEquals("barbar", t2.doubleField());
}

function test3() {
  let t3 = new Obj("baz");
  // hasOwnProperty returns true if the object itself has the field,
  // otherwise it returns false. If the field is on the object's
  // prototype instead (.__proto__), it returns false.
  assertEquals(false, t3.hasOwnProperty("doubleField"));
}
```

Write JavaScript code which will make the above tests pass.

Language Concepts

10.) In regards to memory management, Swift and Python (specifically `cpython`) both use reference counting, whereas Java and JavaScript both use garbage collection.

10.a.) In 1-3 sentences, in your own words, explain how garbage collection reclaims memory. Your description doesn't have to be detailed enough to implement a garbage collector, only detailed enough to get the gist of when memory would be reclaimed.

10.b.) In 1-3 sentences, in your own words, explain how reference counting reclaims memory. Your description doesn't have to be detailed enough to implement reference counting, only detailed enough to get the gist of when memory would be reclaimed.

10.c.) Name one advantage of reference counting over garbage collection.

10.d.) Name one advantage of garbage collection over reference counting.

11.) In 1-3 sentences, explain the difference between compilation and interpretation. Your answer does not need to be detailed enough to implement a compiler or interpreter.

12.) The Java Virtual Machine (JVM) is implemented as an interpreter over Java bytecode. Similarly, most JavaScript implementations are implemented as interpreters. However, most Java and JavaScript implementations support just-in-time (JIT) compilation.

12.a.) In 1-3 sentences, explain what JIT compilation does, in the context of an interpreter. Your answer doesn't need to be detailed enough to implement a JIT compiler.

12.b.) JIT compilers can sometimes generate faster code than traditional compilers. Why?

13.) Swift, Scala, and Haskell all support type inference. In 1-3 sentences, explain what type inference is, and how it relates to statically-typed and dynamically-typed languages. You don't have to provide enough detail to implement a type inferencer.

14.) C only has support for first-order functions, whereas JavaScript and Swift both have support for higher-order functions.

14.a.) In 1-3 sentences, explain what higher-order functions are. You don't have to provide enough detail to explain how to use them.

14.b.) Unlike first-order functions, higher-order functions may require memory to be dynamically allocated at runtime. Why?

14.c.) Write a JavaScript code snippet that uses higher-order functions and would require memory to be dynamically allocated at runtime.

15.) Consider the following code snippet, which is written in some unknown programming language:

```
DefineFunction foo(x, y):  
  DefineVariable temp = x dividedBy y  
  return temp  
  
foo(3, 4)           // first call to foo  
foo("alpha", "beta") // second call to foo
```

15.a.) Assume this language is statically-typed. Does this language probably have type inference? Why or why not?

15.b.) Assume this language is statically-typed. Does this code probably compile? Why or why not?

15.c.) Assume this language is dynamically-typed. Does this code probably compile? Why or why not?

Functions and Higher-Order Functions in Swift

16.) Consider the following code snippet in Swift:

```
let temp = addThree(first: 2, 4, third: 8)
print(temp) // prints 14
```

Implement the `addThree` function below.

17.) Consider the following Swift code:

```
func doOperation(_ f: (Int) -> Int, _ param: Int) -> Int {
    return f(param)
}

let res1 = doOperation({ (x) in x + 5 }, 2)
let res2 = doOperation({ (y) in y - 2 }, 8)
print(res1)
print(res2)
```

What is the output of the above code?

18.) Consider the following Swift code, which calls an unseen function `withAdd`:

```
let addFive = withAdd(amount: 5)
let addSix = withAdd(amount: 6)
print(addFive(2)) // prints 7
print(addFive(5)) // prints 10
print(addSix(2)) // prints 8
print(addSix(3)) // prints 9
```

Implement the `withAdd` function below.

19.) Consider the following Swift code and output, which calls an unseen `doTimes` function:

```
let printFoo = { (x: Int) in print("foo") }
let printCounter = { (y: Int) in print(y) }
let printCounterPlusOne = { (z: Int) in print(z + 1) }
doTimes(printFoo, 2)
print("----")
doTimes(printCounter, 3)
print("----")
doTimes(printCounterPlusOne, 3)
```

Output:

```
foo
foo
----
3
2
1
----
4
3
2
```

If `doTimes'` argument is greater than 0, it will call the passed function, and then recursively call `doTimes` with the argument decremented. Implement `doTimes` below.