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);      // 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.) 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); // calls empty twice
f2(empty); // calls empty three times
```

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

5

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.

```
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));
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();
```

```
10
10

10
15

0
```

0

0
0

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.

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

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`.

```
// 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;
})
```

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.

```
// 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()
});
```

6.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;
})
```

6.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 (accum, element) { return accum + element }, 0)
```

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
```

```
// 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;
}
```

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!
```

```
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!"); }
}
```

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:

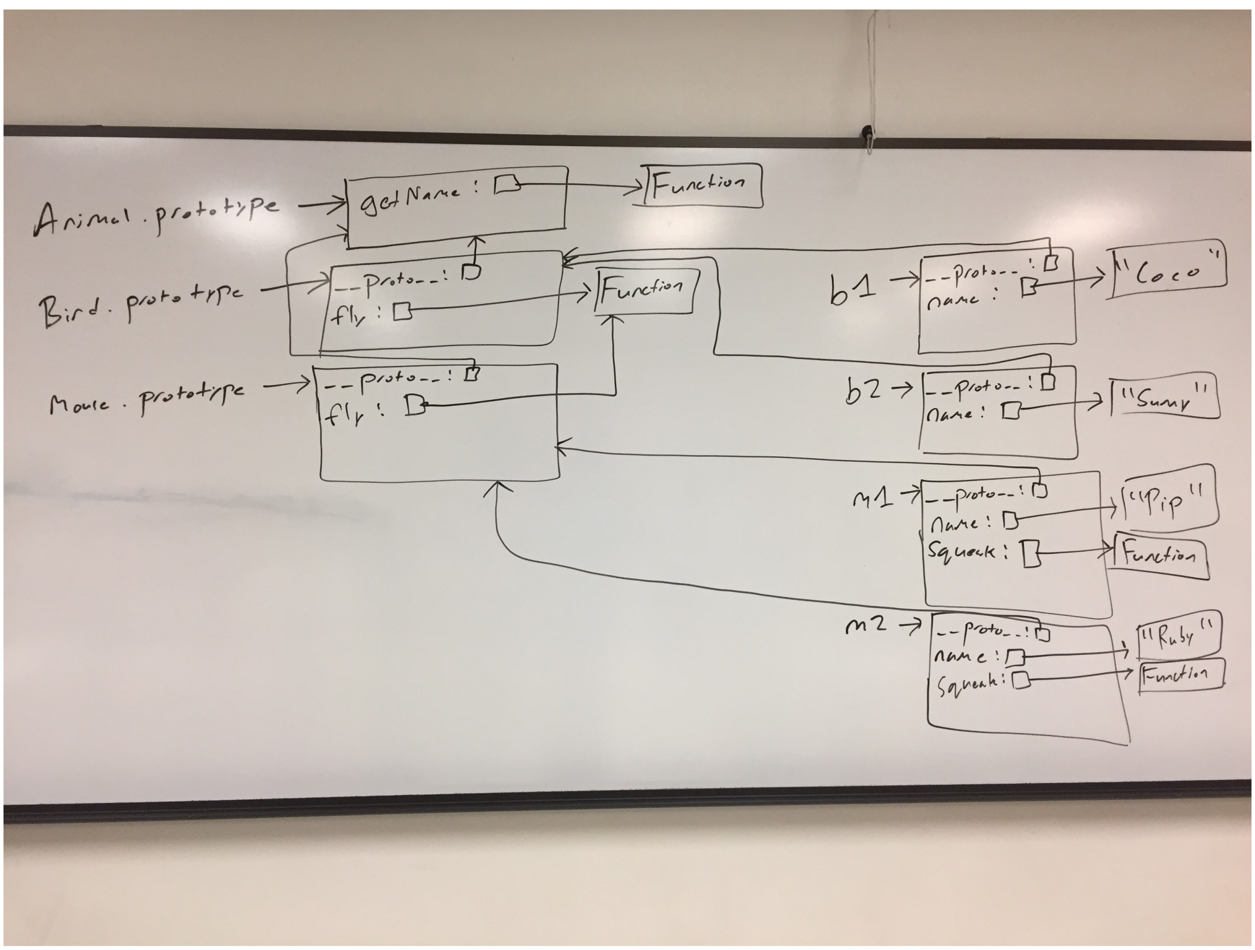
```
let d = new Dog("Rocky");
d.growl(); // prints Rocky growls
```

```
Dog.prototype.growl = function() {
  // assumes constructor initializes this.name, as with 3.a
  console.log(this.name + " 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.

```
// Object is a built-in in JavaScript, but not Obj. This requires a
// custom constructor. From test1, 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 test2, we know that we need a doubleField method on Obj
// objects.  From test3, 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;
}
```

Pattern Matching in Swift

10.) Consider the following `enum` definition:

```
enum SomeEnum {
  case foo(Int)
  case bar(Int, Int)
  case baz(Int, Int, Int)
}
```

Write a function named `test` which takes a value of type `SomeEnum`. The function should do the following:

- If given a `foo`, it should return the value in the `foo`
- If given a `bar`, it should return the sum of the two values in the `bar`
- If given a `baz`, it should return the sum of the **first** and **last** values in the `baz`. You should **not** introduce a variable for the second (middle) value in the `baz`.

An example call to the function follows: `test(SomeEnum.baz(1, 2, 3))`

```
func test(_ value: SomeEnum) -> Int {
  switch value {
    case .foo(let x):
      return x
    case .bar(let x, let y):
      return x + y
    case .baz(let x, _, let y):
      return x + y
  }
}
```

Generics and Higher-Order Functions in Swift

11.) Write the body of the following function, or say if it's impossible to implement. If it's impossible to implement, explain why.

```
func combine<A, B>(a: A, b: B) -> (A, B) {
  return (a, b)
}
```

12.) Write the body of the following function, or say if it's impossible to implement. If it's impossible to implement, explain why.

```
func combine2<A, B>(a: A) -> ((B) -> (A, B)) {
  return { b in (a, b) }
}
```

13.) Write the body of the following function, or say if it's impossible to implement. If it's impossible to implement, explain why.

```
func combine3<A, B>(tup: (A, B)) -> A {
  let (a, _) = tup
  return a
}
```

14.) Write the body of the following function, or say if it's impossible to implement. If it's impossible to implement, explain why.

```
func combine4<A, B>(a: A, f: (A) -> B) -> (A, B) {
  return (a, f(a))
}
```

15.) Consider the following `enum` definition:

```
enum Something<A, B, C> {
  case alpha(A)
  case beta(B)
  case gamma(C)
}
```

15.a.) Write the body of the following function, or say if it's impossible to implement. If it's impossible to implement, explain why.

```
func combine5<A, B, C>(s: Something<A, B, C>) -> (A, B, C) {
  Impossible to implement.  s holds one of an A, B, or C, and the
return type requires all three
}
```

15.b.) Write the body of the following function, or say if it's impossible to implement. If it's impossible to implement, explain why.

```
func combine6<A>(s: Something<A, A, A>) -> A {
  switch s {
    case .alpha(let a): return a
    case .beta(let a): return a
    case .gamma(let a): return a
  }
}
```

16.) Write the body of the following function, or say if it's impossible to implement. If it's impossible to implement, explain why.

```
func combine7<A, B>(f: (A) -> B, b: B) -> A {
  Impossible to implement.  f needs an A, but we only have a B.
}
```

17.) Consider the following enum definition representing lists:

```
indirect enum List<A> {
  case cons(A, List<A>)
  case empty
}
```

17.a.) Write a function named `partition` which takes a predicate and divides a generic list into a pair of returned generic lists. The first element of the pair holds all elements for which the predicate returned `true`, and the second element of the pair holds all elements for which the predicate returned `false`. An example call is below:

```
let (matching, nonmatching) =
  partition(list: List.cons(1, List.cons(2, List.empty)),
            pred: { e in e > 1 })
// matching: List.cons(2, List.empty)
// nonmatching: List.cons(1, List.empty)
```

```
func partition<A>(list: List<A>, pred: (A) -> Bool) -> (List<A>,
List<A>) {
  switch list {
    case .empty: return (List.empty, List.empty)
    case .cons(let head, let tail):
      let (restMatch, restNonMatch) =
        partition(list: tail, pred: pred)
      if pred(head) {
        return (List.cons(head, restMatch), restNonMatch)
      } else {
        return (restMatch, List.cons(head, restNonMatch))
      }
  }
}
```

17.b.) Write a function named `takeWhile` which returns a list of consecutive list elements for which a given predicate `pred` returns `true`. Once `pred` returns `false`, the list is returned. `takeWhile` is generic. Example calls are below:

```
let list = List.cons(1, List.cons(2, List.cons(3, List.empty)))
let first = takeWhile(list: list, pred: { e in e < 3 })
// first: List.cons(1, List.cons(2, List.empty))
let second = takeWhile(list: list, pred: { e in e < 2 })
// second: List.cons(1, List.empty)
let third = takeWhile(list: list, pred: { e in e > 1 })
// third: List.empty
```

```
func takeWhile<A>(list: List<A>, pred: (A) -> Bool) -> List<A> {
  switch list {
    case .cons(let head, let tail):
      if pred(head) {
        return List.cons(head, takeWhile(list: tail, pred: pred))
      } else {
        return List.empty
      }
    case .empty:
      return List.empty
  }
}
```