COMP 333
Fall 2025
Final Practice Exam (Solutions)

The final exam is cumulative. This practice exam, **in addition to** the prior practice exams, assignments, in-class handouts, and exams, is intended to be a comprehensive guide for studying. This practice exam only focuses on material since the last exam. You are permitted to bring three 8.5 x 11 sheets of paper into the exam with you, as long with anything printed or handwritten on them. Both sides of both sheets can be used.

JavaScript

1.) Consider the JavaScript code below and corresponding output, adapted from the second assignment:

```
function Cons(head, tail) {
  this.head = head;
  this.tail = tail;
}
function Nil() {}

let list = new Cons(1, new Cons(2, new Cons(3, new Nil())));
list.forEach((x) => console.log(x));

---OUTPUT---
1
2
3
```

Implement any missing code necessary to produce the above output.

Looking at the above code:

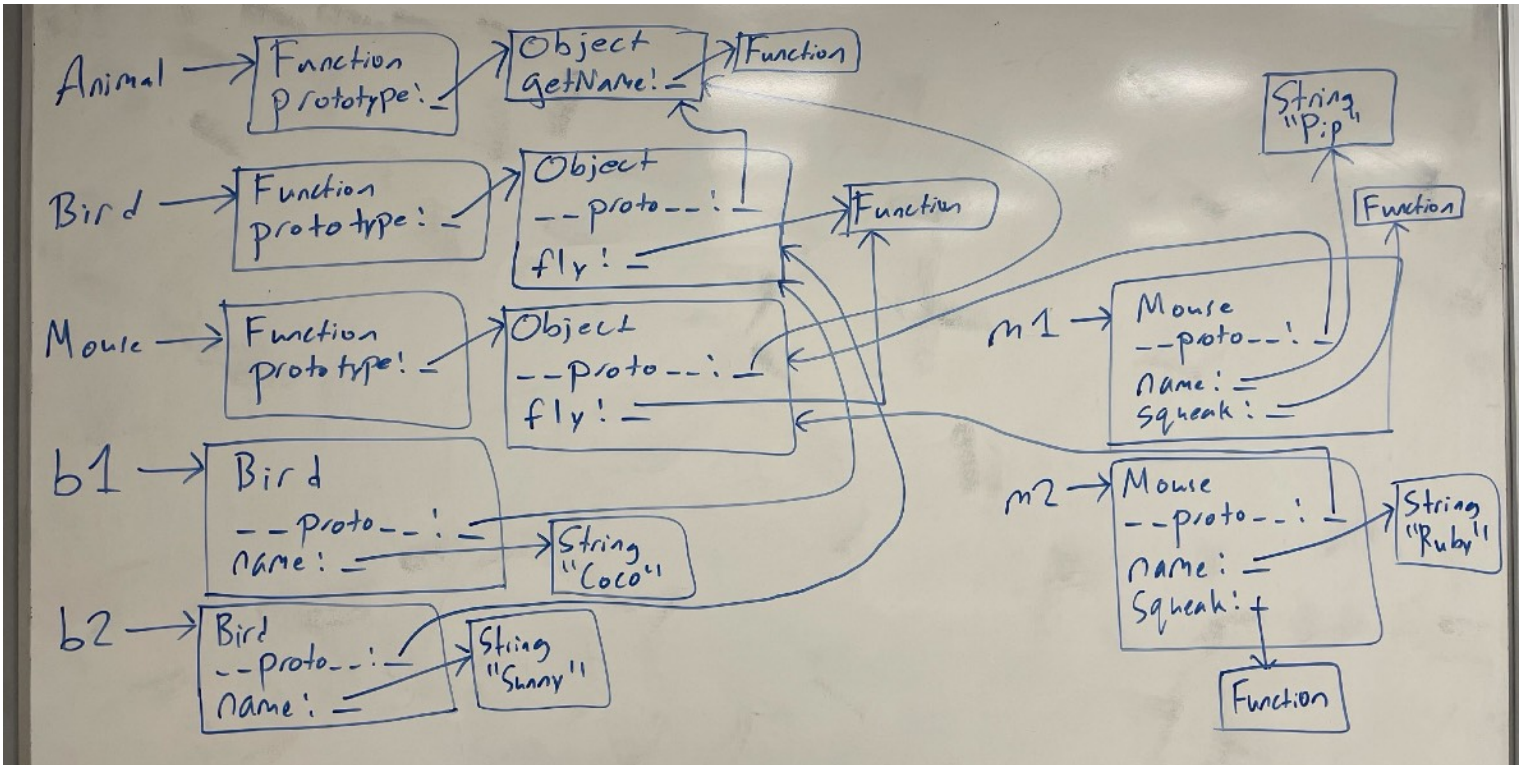
- There must be a `forEach` method defined on lists
  - This takes a parameter: a higher-order function
    - This function itself takes a parameter
    - Appears to be applying the function to each element of the list
  - Does not appear to be returning anything, at least nothing useful
  - Implementation idea: do something different for `Cons` and `Nil`

```
Cons.prototype.forEach = function (f) {
    f(this.head);
    this.tail.forEach(f);
};
Nil.prototype.forEach = function (f) {}
```

2.) 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");
```

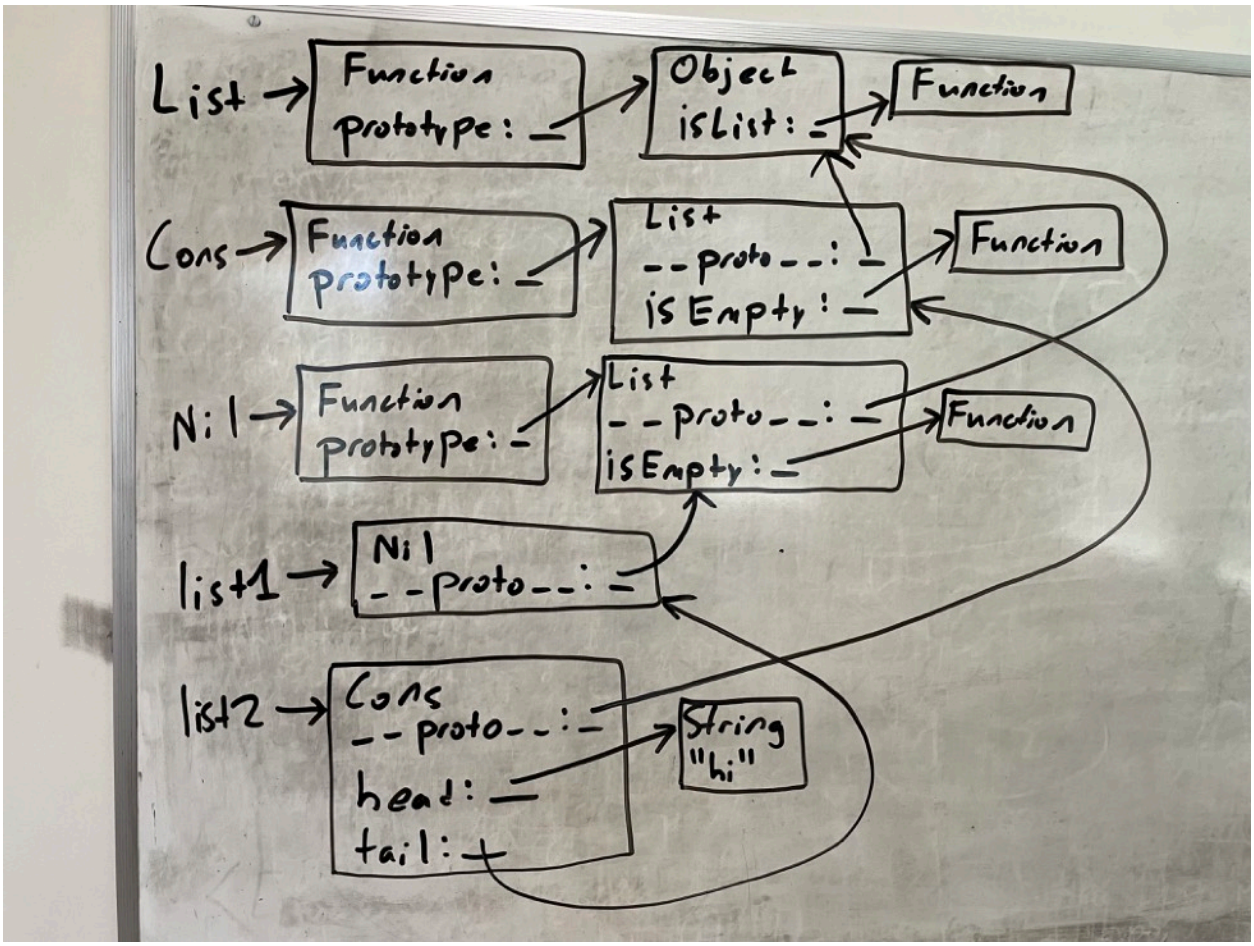
Write a memory diagram which shows how memory looks after this program executes. Your diagram should include the objects and fields associated with `b1`, `b2`, `m1`, `m2`, `Mouse`, `Bird`, and `Animal`. The next page is blank in case you need it.



3.) 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`. The next page is blank in case you need it.



4.) 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;
}
```

5.) 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;
}
```

Memory Management

6.) Java is a garbage-collected language. Consider the following Java code.

```
public static void foo(int x) {
  int a = 3;
  Object b = new Object();
  int c = 4;
  Object d = b;
  Object e = new Object();
}
```

Assume that `foo` is called.

6.a.) What will be allocated on the stack over the duration of `foo`'s call?

All parameters and local variables, namely `x`, `a`, `b`, `c`, `d`, and `e`.

6.b.) What will be allocated on the heap over the duration of `foo`'s call?

Two `Objects`

6.c.) When will the stack-allocated components be deallocated?

When `foo` returns

6.e.) When will the heap-allocated components be deallocated?

Whenever the garbage collector runs

7.) Consider the following Java code:

```
public class Foo {
  public int x;
  public Foo(int y) {
    x = y;
  }

  public static void main(String[] args) {
    Foo obj = new Foo(1);
  }
}
```

For each variable in the program, list whether the variable will be allocated on the stack or the heap.

Stack-allocated: `y`, `args`, `obj`. These are all either local variables or method parameters.

Heap-allocated: `x`. Instance variables get allocated on the heap, specifically when the object is created (`new Foo(1)` in the code above).

8.) C requires users to explicitly allocate and deallocate heap memory. This is in contrast to automated memory management techniques, wherein deallocation is performed automatically.

8.a.) Name one advantage of explicit deallocation over automated deallocation.

Possible answers, among others:
- More flexible; users have complete control over memory
- Automated techniques either impart potentially significant overhead (e.g., garbage collection, reference counting), or require programmers to follow strict restrictions (e.g., ownership and borrowing)

8.b.) Name one advantage of automated deallocation over explicit deallocation.

Possible answers, among others:
- No need to worry about when memory will be freed
- Prevents bugs related to freeing memory too early (e.g., use after free), too late, or not at all (i.e., memory leaks)

9.) What is wrong with the following C code performing explicit memory management?

```
void foo() {
  int* p = malloc(sizeof(int));
  *p = 5;
}
```

The memory allocated by `malloc` is never freed, leaking memory each time `foo` is called.

10.) What is wrong with the following C code performing explicit memory management?

```
void foo() {
  int* p = malloc(sizeof(int));
  *p = 5;
  free(p);
  *p = 6;
}
```

The memory deallocated by `free` is used after it was deallocated. This is a use-after-free bug, and it's undefined behavior in C.

11.) Programs generally can dynamically allocate memory in one of two places: the stack and the heap.

11.a.) Name one advantage of stack allocation over heap allocation.

Possible answers, among others:

- Completely automatic allocation and deallocation
- Allocation and deallocation are computationally cheap - pushing/popping values off of a stack
- Corresponds well to how functions are called/return; push onto the stack when a function is called, and pop when it returns

11.b.) Name one advantage of heap allocation over stack allocation.

Possible answers, among others:
- Does not require allocated data to be copied upon calling a function or returning from a function
- The size of the allocated item does not need to be known at compile time
- Well-suited for allocated items that need to remain in memory across many function calls

12.) Consider the following Java function:

```
public static void foo() {
  int x = 0;
  Object obj1 = new Object();
  Object obj2 = obj1;
}
```

12.a.) If `foo` is called, what is allocated on the stack?

`x`, `obj1`, `obj2`

12.b.) If `foo` is called, what is allocated on the heap?

a single `Object`

12.c.) When `foo` returns, what is guaranteed to be deallocated immediately?

Anything allocated on the stack, namely `x`, `obj1`, and `obj2`

12.d.) When `foo` returns, is there anything which might be deallocated at a later point? That is, what is *not* guaranteed to be immediately deallocated?

Anything allocated on the heap, namely the single `Object`