# COMP 333 Final Practice Exam

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.

## Language Terminology

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

1.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.

Starting from all variables declared on the stack (or globally), we trace through memory and determine which parts of memory are reachable or not. The parts of memory which are not reachable are reclaimed.

1.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.

Each allocated chuck of memory is given a reference count, which is incremented whenever a new reference is made to this memory, or decremented whenever an existing reference disappears. If the reference count hits zero, the memory is reclaimed.

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

Possible answers, among others:

- Memory is generally reclaimed as soon as it is no longer needed
- Happens at much more predictable times than garbage collection
- Application performance tends to be much more consistent than with garbage collection

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

Possible answers, among others:
- Can reclaim cyclic data structures without programmer intervention
- Generally better performance in terms of lower overall execution times

2.) 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.

Compilers translate programs written in one programming language to another, whereas interpreters directly execute programs written in a given programming language.

3.) 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.

3.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.

The interpreter dynamically monitors code and can choose to compile chunks of frequently-executed code directly to machine code. When these chunks are executed in the future, it can jump to the compiled chunks of machine code as opposed to interpreting the corresponding original code.

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

Traditional compilers can only reason about how code *might* execute, whereas JIT compilers can observe exactly how code *is* executed. As such, the JIT compiler has more information to work with, and can use this to generate faster, more optimal code.

4.) 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.

Type inference allows a compiler for a statically-typed language to infer what the types of variables must be, without the programmer explicitly saying what the types are. The resulting code may look like dynamically-typed code (because it lacks type information), but it's still statically-typed.

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

5.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.

Higher-order functions allow for functions to be created dynamically at runtime. As part of this, functions become another kind of data, so they can be saved in variables, passed to other (possibly higher-order) functions, and returned from (possibly higher-order) functions.

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

They might "close-over" a value from an enclosing scope. For example, consider the following JavaScript snippet:

```
function foo(a) {
  return function (b) {
    return a === b;
  }
}
```

The function returned by `foo` needs to save `a` somewhere, and `foo` could be called an arbitrary number of times. As such, dynamic memory allocation is necessary.

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

The snippet from 5.b. is an example of one such function.

6.) 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
```

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

Yes, because no types are explicitly written in the code. As such, it's probably either dynamically-typed, or statically-typed with type inference.

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

Probably not. foo seems to be doing division, but it's called with both integers and strings. Division doesn't make sense for strings.

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

It probably compiles, even though there will likely be a runtime error at the second call to `foo`.

## Swift

7.) Consider the following incomplete Swift code and output:

```
let sum: Int = add(first: 3, 4);
let product: Double = multiply(2.3, second: 6.5);
print(sum);
print(product);

---OUTPUT---
7
8.8
```

Write out any missing code below that would allow this code to compile with the correct output.

```
func add(first: Int, _ second: Int) -> Int {
    return first + second;
}

func multiply(_ first: Double, second: Double) -> Double {
    return first * second;
}
```

8.) Consider the following incomplete Swift code and output:

```
let d1: SomeData = SomeData.foo(2, 3.1);
let d2: SomeData = SomeData.bar(true);
let d3: SomeData = SomeData.baz;

print(d1);
print(d2);
print(d3);


---OUTPUT---
foo(2, 3.1)
bar(true)
baz
```

Write out any missing code below that would allow this code to compile with the correct output.

```
enum SomeData {
    case foo(Int, Double)
    case bar(Bool)
    case baz
}
```

9.) Consider the following incomplete Swift code and output, which calls an unseen function named `take`:

```
indirect enum List {
    case Nil
    case Cons(String, List)
}

let list =
  List.Cons("foo",
    List.Cons("bar",
      List.Cons("baz",
        List.Nil)));

print(take(list, -1));
print(take(list, 0));
print(take(list, 1));
print(take(list, 2));
print(take(list, 3));
print(take(list, 4));


---OUTPUT---
Nil
Nil
Cons("foo", List.Nil)
Cons("foo", List.Cons("bar", List.Nil))
Cons("foo", List.Cons("bar", List.Cons("baz", List.Nil)))
Cons("foo", List.Cons("bar", List.Cons("baz", List.Nil)))
```

The `take` function takes a list and a number of elements `n`, and it will return a new list holding the first `n` elements of the original list. If `n <= 0`, then the empty list is returned. If `n >` the length of the list, then the entire list is returned (though possibly a copy of the original list). Implement `take` below. The next page is blank in case it is needed.

```
func take(_ list: List, _ n: Int) -> List {
    switch list {
    case .Nil:
        return .Nil;
    case let .Cons(head, tail):
        if n <= 0 {
            return List.Nil;
        } else {
            return List.Cons(head, take(tail, n - 1));
        }
    }
}
```

10.) Consider the following incomplete Swift code and output that calls an unseen `bothTrue` function:

```
let lessThanThree: (Int) -> Bool = { i in i < 3 };
let isEven: (Int) -> Bool = { x in x % 2 == 0 };
let greaterThanTwo: (Int) -> Bool = { i in i > 2 };
let isOdd: (Int) -> Bool = { i in i % 2 == 1 };

print(bothTrue(lessThanThree, isEven, 2));
print(bothTrue(greaterThanTwo, isOdd, 3));
print(bothTrue(isOdd, isEven, 4));
print(bothTrue(lessThanThree, isOdd, 1));

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

Implement `bothTrue` below.

```
func bothTrue(
  _ f1: (Int) -> Bool,
  _ f2: (Int) -> Bool,
  _ param: Int) -> Bool {
    return f1(param) && f2(param);
}
```

11.) Consider the following Swift code, that makes use of type inference:

```
let p1 = 5;
let p2 = true;
let p3 = { x in x + p1 };
let p4 = { y in p2 || y };
let p5 = { z in p1 > z };
let p6 = { (a, b) in a + b + p1 };
```

Record the types of `p1` through `p6` below.

```
p1: Int
p2: Bool
p3: (Int) -> Int
p4: (Bool) -> Bool
p5: (Int) -> Bool
p6: (Int, Int) -> Int
```

12.) Consider the following Swift code, which uses function overloading to redefine `foo` twice with different types:

```
func foo(_ x: String, _ y: Int) -> (String, Int) {
    return (x, y);
}

func foo(_ x: Bool, _ y: Double) -> (Bool, Double) {
    return (x, y);
}
```

Redefine `foo` below to work with any two types. As a hint, you will need to add type variables.

```
func foo<A, B>(_ x: A, _ y: B) -> (A, B) {
    return (x, y);
}
```

13.) Consider the following Swift code, which duplicates the definition of a list and `length` in order to handle different types.

```
indirect enum IntList {
    case IntNil
    case IntCons(Int, IntList)
}

func length(list: IntList) -> Int {
    switch list {
    case .IntNil:
        return 0;
    case let .IntCons(_, tail):
        return 1 + length(list: tail);
    }
}

indirect enum StringList {
    case StringNil
    case StringCons(String, StringList)
}

func length(list: StringList) -> Int {
    switch list {
    case .StringNil:
        return 0;
    case let .StringCons(_, tail):
        return 1 + length(list: tail);
    }
}
```

Rewrite the code below to remove the duplication via the use of type variables. Your rewritten code should only contain one `enum` definition and one `length` function. You may rename the lists with whatever type names and constructor names you want. The next page is intentionally blank to give you more room.

```
indirect enum List<A> {
    case Nil
    case Cons(A, List<A>)
}

func length<T>(list: List<T>) -> Int {
    switch list {
    case .Nil:
        return 0;
    case let .Cons(_, tail):
        return 1 + length(list: tail);
    }
}
```