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.

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.

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

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

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.

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.

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

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.

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.

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

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

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?

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

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

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.

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.

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.

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.

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.

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.

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.