COMP 333 Final Practice Exam

The final exam is cumulative. This practice exam, **in addition to** the prior practice exams, assignments, and in-class handouts, 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.) 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)




}
```

8.) 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) }




}
```

9.) 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




}
```

10.) 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))




}
```

11.) Consider the following `enum` definition:

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

11.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




}
```

11.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
  }









}
```

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









}
```

13.a.) Define a protocol named `Equals` which defines an `equals` method. `equals` returns `true` if two values equal each other. Example calls are below, assuming `Int` implements the `Equals` protocol:

```
1.equals(1) // returns true
2.equals(3) // returns false
```

```
protocol Equals {
  func equals(_ other: Self) -> Bool
}
```

13.b.) Implement the `Equals` protocol for `Int`, using `extension`. As a hint, `==` can be used to test if two integers are identical, as with `1 == 2`.

```
extension Int: Equals {
  func equals(_ other: Int) -> Bool {
    return self == other
  }
}
```

13.c.) Consider the following enum definition:

```
enum Thing<A> {
  case thing1
  case thing2(A)
}
```

Implement the `Equals` protocol for `Thing`, using `extension`. As a hint, you'll need to tell the compiler that `A` needs to implement the `Equals` protocol, and you can use a `where` clause for this purpose. Two `thing1` values should equal each other, and a `thing2` value should equal another `thing2` value if both `thing2` values contain the same value of type `A`.

```
extension Thing : Equals where A : Equals {
  func equals(_ other: Thing<A>) -> Bool {
    switch (self, other) {
      case (.thing1, .thing1):
        return true
      case let (.thing2(leftThing), .thing2(rightThing)):
        return leftThing.equals(rightThing)
      case _:
        return false
    }
  }
}
```

14.) The following code does not compile. Why not?

```
protocol Foo {
  func fooMethod() -> Bool
}
extension Int : Foo {
  func fooMethod() -> Bool {
    return true
  }
}
true.fooMethod()
```

`Bool` does not implement the `Foo` protocol, and the `Foo` protocol is needed to call `fooMethod`.

Prolog

Basic Procedures

15.) Define a procedure named `isFish` which encompasses the following idea: `goldfish`, `bass`, and `carp` are all fish. `isFish` should be defined as a series of facts.

```
isFish(goldfish).
isFish(bass).
isFish(carp).
```

16.) Assume the presence of a procedure named `isInstrument`, which lists various musical instruments. Define a procedure named `musicalFish`, which succeeds if the input is both a fish (according to `isFish`) and an instrument (according to `isInstrument`; `bass` are both).

```
musicalFish(Input) :-
  isFish(Input),
  isInstrument(Input).
```

17.) Consider the following procedure:

```
foo(0).
foo(1) :-
  X = 1.
foo(2) :-
  X = Y,
  X = 1,
  Y = 2.
foo(3).
```

What are the solutions to the following query?

```
?- foo(X).

X = 0 ;
X = 1 ;
X = 3 .
```

18.) Consider the following procedure showed in class, which computes Fibonacci numbers:

```
fib(0, 0).
fib(1, 1).
fib(N, Result) :-
  N > 1,
  MinOne is N - 1,
  MinTwo is N - 2,
  fib(MinOne, Temp1),
  fib(MinTwo, Temp2),
  Result is Temp1 + Temp2.
```

18.a.) Why is the `N > 1` clause necessary in the recursive case?

To ensure mutual exclusion with the base cases. Otherwise, this procedure could get stuck in infinite recursion trying to compute negative Fibonacci numbers.

18.b.) Consider the following rewritten version of this code, which is intended to be more concise:

```
fib(0, 0).
fib(1, 1).
fib(N, Result) :-
  N > 1,
  fib(N - 1, Temp1),
  fib(N - 2, Temp2),
  Result = Temp1 + Temp2.
```

This code doesn't work properly. Why not?

is must be used to perform any arithmetic. To illustrate what happens with this rewritten version, the recursive case is internally parsed as the following:

```
fib(N, Result) :-
  N > 1,
  fib(-(N, 1), Temp1),
  fib(-(N, 2), Temp2),
  Result = +(Temp1, Temp2).
```

That is, the arithmetic operations are actually parsed as structures; `N - 1` becomes `-(N, 1)`, a structure named `-` with parameters `N` and `1`.