COMP 333
Summer 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 as they have handwritten notes on them. Both sides of both sheets can be used. To be clear, these must be entirely handwritten.

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) {}
```

Language Concepts

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.

Memory Management

4.) 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.

4.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`.

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

Two `Objects`

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

When `foo` returns

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

Whenever the garbage collector runs

5.) 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).

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

6.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)

6.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)

7.) 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.

8.) 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.

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

```
void foo() {
  int* p1 = malloc(sizeof(int));
  int* p2 = p1;
  free(p1);
  free(p2);
}
```

The same chunk of memory was freed twice, referred to as a double-free. This has undefined behavior in C. Heap-allocated memory can be freed at most one time.

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

```
int* foo() {
  int* p1 = malloc(sizeof(int));
  free(p1);
  return p1;
}
```

Technically nothing, but its so strange that it is likely a bug. `p1` no longer refers to valid memory after the call to `free`. However, `foo` then returns `p1`. This is not undefined behavior, but trying to access the memory `p1` points to *would be* undefined behavior. With this in mind, there is almost no reason why anyone would want to use `foo`'s return value.

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.) There exist multiple automated memory management techniques. For example, Java and JavaScript both use garbage collection, Python and Swift use reference counting, and Rust uses ownership and borrowing.

12.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. This tracing and deallocation is performed at runtime.

12.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.

12.c.) In 1-3 sentences, in your own words, explain how Rust's ownership and borrowing system reclaims memory. It doesn't need to be detailed enough to implement it, only detailed enough to know when memory would be reclaimed.

Every bit of allocated memory has a single unique owner, which are traced all the way back to variables declared on the stack. When a stack variable is deallocated, any memory owned by the variable is deallocated. The deallocation itself is performed at runtime, but we can determine when these deallocations happen at compile time.

12.d.) 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

12.e.) Name one advantage of garbage collection over reference counting.

Possible answers, among others:
- Can handle cyclic data structures
- Advanced garbage collectors can more easily reposition memory, perform optimizations, and generally are more flexible in their capabilities than reference counting

12.f.) Name one advantage of Rust's ownership/borrowing system over garbage collection.

Possible answers, among others:
- Memory is reclaimed as soon as it is no longer needed
    - Likely still in a cache, so reclamation will likely be faster
    - Less wasted memory overall
- Predictable behavior at runtime, making it appropriate for real-time systems
- No runtime component for memory management - less overhead

12.g.) Name one advantage of garbage collection over Rust's ownership/borrowing system.

Possible answers, among others:
- Garbage collection is fully automatic, and you generally don't need to think about it
- Can have many references to the same object without restrictions (e.g., you don't have to designate any of them as the owner, nor can you)
- Overall more flexible
- More popular - better understood by more people

13.) C only has support for first-order functions, whereas JavaScript has support for higher-order functions.

13.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.

13.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.

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

The snippet from 13.b is an example of one such function.

14.) Consider the following Java function:

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

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

`x`, `obj1`, `obj2`

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

a single `Object`

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

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

14.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`

Rust

15.) Declare a struct named `Example` that holds five fields:

- `first`, which holds a `String`
- `second`, which holds a 32-bit unsigned integer
- `third`, which holds a 32-bit signed integer
- `fourth`, which holds a 64-bit unsigned integer
- `fifth`, which holds a 64-bit signed integer

```
struct Example {
  first: String,
  second: u32,
  third: i32,
  fourth: u64,
  fifth: i64
}
```

16.) Consider the following Rust code:

```
fn main() {
    let s1: String = "foo".to_string();
    let s2: String = s1;
    println!("{}", s1);
}
```

This code does not compile, and the compiler produces an error message pointing to the use of `s1` in the `println!` statement. Why doesn't this code compile?

Ownership of the `String` was transferred from `s1` to `s2`, so `s1` can no longer access the `String`. However, this program still attempted to access the `String` through `s1`.