**COMP 333**
**Fall 2026**
**Midterm Practice Exam #2**

This is representative of the kinds of topics and kind of questions you may be asked on the midterm. This practice exam, alone with assignment one and the in-class handout on types, is intended to be comprehensive of everything on the exam. That is, I will not ask anything that's not somehow covered by those sources.

You are permitted to bring two 8.5 x 11 sheets of paper into the exam with you, containing either printed or handwritten notes. Both sides of both sheets can be used.

## Types

1.) Name one reason why someone might prefer static typing over dynamic typing.

2.) Name one reason why someone might prefer dynamic typing over static typing.

3.) Name one reason why someone might prefer strong typing over weak typing.

4.) Name one reason why someone might prefer weak typing over strong typing.

5.) Consider the following code, written in some unknown language:

```
define doSomething(x) {
  return x.foo(7);
}

obj = Foo("hello")
value = doSomething(obj)
print(value)
obj = Foo("goodbye")
```

Provide an argument why this language might be statically-typed, OR why it might be dynamically-typed. Both are possible; the explanation why is the only important part.

6.) In your own words, what is undefined behavior?

7.) Consider the following code, written in some unknown language:

```
x = 5
y = true
x = false
```

Provide an argument why this language might be statically-typed, OR why it might be dynamically-typed. Both are possible; the explanation why is the only important part.

8.) Code written in type-inferred languages can look a lot like dynamically-typed code, but it nonetheless has the same performance advantages as statically-typed code. How?

9.) Assembly languages are typically untyped. What does "untyped" mean, and how does this differ from dynamic typing?

10.) Consider the following code snippet in some unknown object-oriented language:

```
Object obj = new Foo();
Foo f = (Foo)obj;
```

The following checks are relevant:

1. Ensure that class `Object` and `Foo` exist
2. Ensure that `Foo`'s constructor takes no arguments
3. Ensure that `obj` is in scope on the second line
4. With respect to the cast on the second line, ensure that `obj`'s type is actually `Foo`

Identify which checks are most likely to be performed when for each kind of language.

| | **Statically-Typed** | **Dynamically-Typed** |
|---|---|---|
| **Strongly-Typed** | Runtime:<br><br>Compile-time: | Runtime:<br><br>Compile-time: |
| **Weakly-Typed** | Runtime:<br><br>Compile-time: | |

11.) The code below is written in some unknown language. The code attempts to access an array at some index, and then print the value that was at that position:

```
let element = arr[index];
print(element);
```

You run this code snippet with different values for `arr` and `index`, including for values of `index` which are out of bounds for `arr`. With out-of-bounds array accesses, you observe that the code prints out a variety of values, and even sometimes crashes.

From these observations, is this language most likely strongly-typed or weakly-typed? In 1-2 sentences, explain your reasoning.

12.) In C, it is undefined behavior to attempt to access an array out of bounds. Consider the following C program which creates an array containing 3 elements, and attempts to access it from different indices. Assume that the `print_string` function prints a provided string, and the `print_int` function prints a provided integer.

```
print_string("start");
int arr[3] = { 7, 8, 9 };
print_int(arr[0]); // access in bounds
print_int(arr[3]); // access out of bounds - undefined behavior
```

You attempt to compile this program, and it compiles without warnings or errors. You then run this program, and observe it produces the following output:

```
start
7
0
```

12.a.) If you run the program again, would you expect it to produce the same output? Why or why not? Either answer is possible; your explanation is the important part.

12.b.) Why didn't we get any sort of indicator that we triggered undefined behavior?

12.c.) What parts of this output, if any, are guaranteed to always be the same across different runs of this program?

**Assignment 1**

13.) Consider the following code implementing an immutable linked list of **`booleans`** (based on assignment #1). `Example.java` contains `main`, and the expected output is shown in the comments.

```
public interface ImmutableList {
    public boolean orAll();
}

public class Cons implements ImmutableList {
  public final boolean head;
  public final ImmutableList tail;
  public Cons(final boolean head, final ImmutableList tail) {
    this.head = head;
    this.tail = tail;
  }
  public boolean orAll() { ... }
}

public class Nil implements ImmutableList {
  public Nil() {}
  public boolean orAll() { ... }
}

public class Example {
  public static void main(String[] args) {
    ImmutableList empty = new Nil();
    System.out.println(empty.orAll()); // prints false
    ImmutableList trueFalse =
      new Cons(true, new Cons(false, empty));
    System.out.println(trueFalse.orAll()); // prints true
    ImmutableList falseFalse =
      new Cons(false, new Cons(false, empty));
    System.out.println(falseFalse.orAll()); // prints false
  }
}
```

`orAll` computes logical OR (`||`) over all list elements, with `Nil` considered as `false`. Implement `orAll` for `Cons` and `Nil` on the next page. Unlike assignment 1, you may implement this however you want (i.e., loops, `if`, and ternary are ok, but not necessary). The comments on the next page indicate whether the method is for `Cons` or `Nil`.

```
// Nil
public boolean orAll() {









}
```

```
// Cons
public boolean orAll() {

















}
```

14.) Consider the following code implementing an immutable linked list of `int`s (based on assignment #1):

```
public interface IList {
  public IList copy();
}

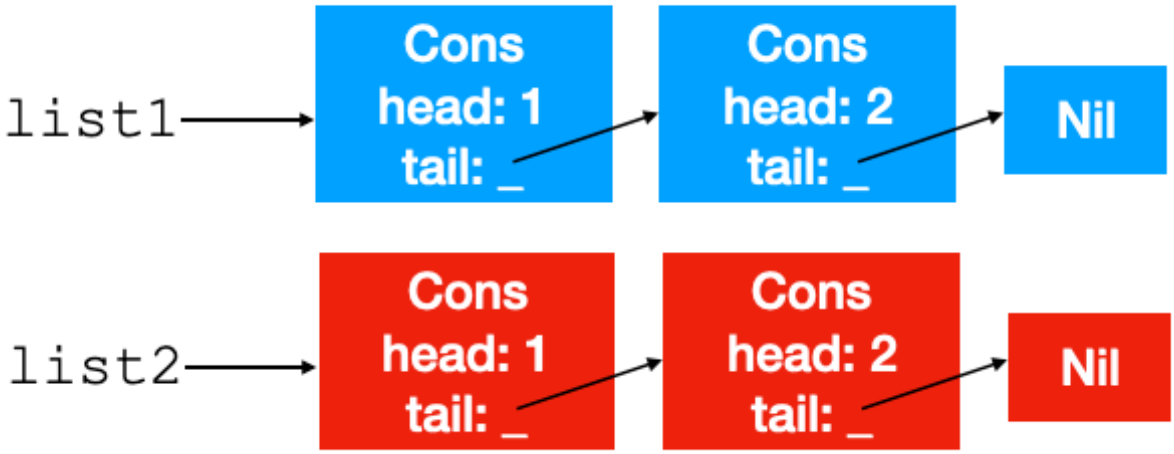
public class Cons implements IList {
  public final int head;
  public final IList tail;
  public Cons(final int head, final IList tail) {
    this.head = head;
    this.tail = tail;
  }
  public IList copy() { ... }
}

public class Nil implements IList {
  public Nil() {}
  public IList copy() { ... }
}
```

The `copy` method is intended to create a deep copy of the given list, meaning a whole new list will be allocated in memory, with no references into the existing list. For example, consider the following code:

```
IList list1 = new Cons(1, new Cons(2, new Nil()));
IList list2 = list1.copy();
```

This code would produce the following in memory:




On the next page, implement your copy method for `Cons` and `Nil`.

```
// Nil
public IList copy() {









}
```

```
// Cons
public IList copy() {

}
```

15.) The following is a lightly modified version of `length` that we did in class:

```
public interface IList {
  public int length();
}

public class Nil implements IList {
  public Nil() {}
  public int length() {
    return 0;
  }
}

public class Cons implements IList {
  public final int head;
  public final IList tail;
  public Cons(int head, IList tail) {
    this.head = head;
    this.tail = tail;
  }
  public int length() {
    return 1 + tail.length();
  }
}
```

As shown, `Cons`' definition of `length` is implemented recursively. However, there is no use of `if` or the ternary operator. Why is this not infinitely recursive?

16.) Mutable linked lists usually don't expose nodes in their interfaces, since it would be easy for someone to make unintended modifications and corrupt the list. However, with immutable lists, users directly hold nodes, and different lists may even share nodes. Why aren't nodes similarly protected with immutable lists?