COMP 333
Fall 2026
Midterm Practice Exam #2 (Solutions)

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.

Possible answers:
- Can help eliminate programming errors at compile time
- Can help in structuring code (types can be a guide for program development)
- Compilers and runtimes can usually optimize code better with type information

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

Possible answers:
- No need to annotate programs with types, which are a significant amount of code
- More programs are possible, and can fundamentally represent things which cannot be represented in statically-typed code.

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

Possible answers:
- Strongly-typed languages are more predictable, especially in the presence of bugs and errors
- Strongly-typed languages tend to emphasize code correctness

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

Possible answers:

- There is usually a performance benefit. For example, there is no need for runtime array bounds checking.
- Weakly-typed languages tend to be more expressive. (Beyond what I'd expect: C lets you do more than Java in terms of low-level manipulation, but it's easy to hit undefined behavior, meaning programs lose all meaning. This can happen even when the code appears to be working correctly.)

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.

Reasoning for dynamic typing: no types are present in the code, consistent with any dynamically-typed programming language.

Reasoning for static typing: none of the variables ever appear to change type, and `doSomething` always returns something of the same type, as long as `foo` always returns something of the same type. This is consistent with static typing with type inference, where the types are determined at compile-time, but the programmer never needs to explicitly write the types.

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

One possible definition: a condition under which a program, according to the language's definition, loses all meaning. It's a programmer's responsibility to avoid 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.

Reasoning for dynamic typing: no types are present in the code, consistent with dynamic typing. Additionally, assuming that `5` and `false` are of different types, `x` seems to change its type here.

Reasoning for static typing: this could be type inferred, which is still statically-typed, though this is admittedly a lot less likely. While x appears to change its type, this could be that either:

1. `5` and `false` are actually of the same type
2. `5` and `false` are subtypes of some other type `T`, and the compiler infers `x` to be of type `T`

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?

Type inference runs at compile time, not runtime, so we don't pay a runtime cost for type inference. The compiled version of the code ends up being the same as a version with explicit types, it's just that the compiler will figure out those types for us.

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

Untyped: neither variables nor values have types associated with them. Data is a bunch of untagged bits with no inherent meaning. If meaning is desired, programmers are responsible for maintaining this themselves.

With dynamic typing, values carry self-descriptive tags at runtime saying what the type of the value is. This, for example, is how JavaScript knows what the intention behind the expression `x + y` is; it will look at the tags stored on the values in `x` and `y`, and decide between numeric addition and string concatenation depending on these tags.

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: 4 Compile-time: 1, 2, 3	Runtime: 1, 2, 3, 4 Compile-time:
Weakly-Typed	Runtime: Compile-time: 1, 2, 3	

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.

Weakly-typed. The behavior is unpredictable, indicating that no bounds check appears to be happening. Weakly-typed languages avoid runtime checks like bounds checks.

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.

Why: `arr[3]` is the only indeterminate value printed, since it's outside of the array. We saw large random-looking integers in class when numbers changed, and small numbers when they didn't change. 0 isn't guaranteed, but it's a little *too* non-random.

Why not: `arr[3]` triggers undefined behavior, meaning there is absolutely no guarantee as to what will happen. The program is free to do whatever it wants here.

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

It's up to programmers to avoid undefined behavior, and programs may assume the programmer never triggers undefined behavior. Automatically checking for all kinds of undefined behavior leads to costly runtime checks, and weakly-typed languages with undefined generally want to run as quickly as possible.

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

Confusingly, none. While the last line is the only line that directly triggers undefined behavior, the C standard says that once we start going down a code path that is guaranteed to eventually trigger undefined behavior, then the *whole path* becomes undefined behavior. In other words, because this program will always attempt to access an array out of bounds, the *whole program* has no fixed meaning, so none of the prints are guaranteed to happen.

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() {
  return false;









}
```

```
// Cons
public boolean orAll() {
  return head || tail.orAll();









}
```

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

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

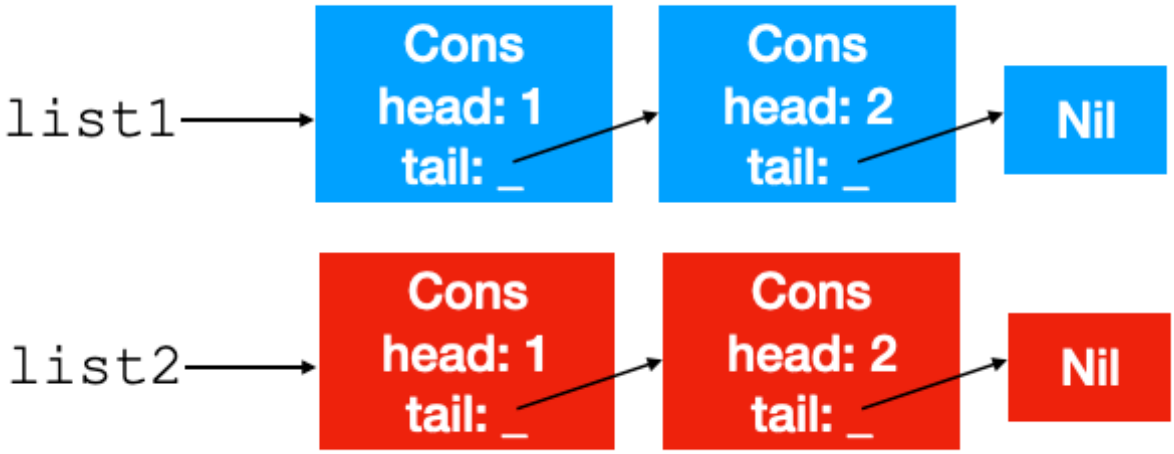
public class Cons implements IList {
  public final int head;
  public final IList tail;
  public Cons(final int head, final IList tail) {
    this.head = head;
    this.tail = tail;
  }
  public IList copy() { ... }
}

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() {
  return new Nil();
}

// Cons
public IList copy() {
  return new Cons(head, tail.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?

Since `Cons`' tail instance variable has the compile-time type of `IList`, its runtime type could be either `Nil` or `Cons`. If the runtime type is `Nil`, then the call to `tail.length()` will call `Nil`'s definition of `length`, i.e., the base case, terminating the recursion. If the runtime type is `Cons`, then the call to `tail.length()` will recursively call `Cons`' length method. We therefore effectively get conditional execution through Java's ad-hoc polymorphism.

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?

Since immutable lists can't be modified, there is chance of corruption due to unintended modifications; you can't have unintended modifications if no modifications are possible at all. This is also why sharing nodes is safe: it's guaranteed that nothing ever changes.