1. Covariant and contravariant type parameters

In Scala, a type parameter in a definition can be marked covariant by prefixing it with + and contravariant by prefixing it with −.

Consider the following Scala code:

```scala
abstract class Super

class Sub1(n: Int) extends Super

class Sub2(b: Boolean) extends Super

class Box1[+A] // covariant

class Box2[−A] // contravariant

def g1(x: Box1[Super]) = x

def g2(x: Box1[Sub1]) = x

def h1(x: Box2[Super]) = x

def h2(x: Box2[Sub1]) = x
```

Suppose that A is replaced with any of Super, Sub1, or Sub2. For which A do the following calls typecheck:

- `g1(new Box1[A])`
- `g2(new Box1[A])`
- `h1(new Box2[Super])`
- `h2(new Box2[Sub1])`

(It may help to draw a matrix with rows labeled by the function names and columns by the three possible types for A). You can type all of these expressions into Scala to find this out. What is the pattern?

2. Parameterized traits

Traits can also be parameterized by types. The builtin trait `Ordered[T]` is an example:

```scala
trait Ordered[T] {
  def compare(that: T): Int
  def < (that: T): Boolean = ???
  def <= (that: T): Boolean = ???
}
```

Here, the type parameter T is needed to name the type of other elements to which this will be compared. The `this.compare(that)` operation returns a
negative integer if this is less than that, zero if they are equal and a positive integer if this is greater than that.

Based on this specification, fill in the ??? regions in the above code snippet with code that defines standard comparison operators such as < in terms of compare. Define the remaining operations >, >=, != ==.

3. List comprehensions

Using the desugaring rules for list comprehensions described in Lecture 12, give the resulting list and convert the following list comprehension expressions to plain Scala code.

(a) for (x <- List(1,2,3)) yield (x + 1)
(b) for (x <- List(1,2,3); if (x % 2 == 0)) yield (x / 2)
(c) for (x <- List(1,2,3); y <- List(1,2,3); if (x < y)) yield (x, y)

4. Covariant lists

Covariance and subtyping allows us to define lists more cleanly:

```scala
abstract class List[+A]
case object Nil extends List[Nothing]
case class Cons[A](head: A, tail: List[A]) extends List[A]
```

Now Nil is a case object, so it doesn’t need a parameter list, and it extends List[Nothing], so it doesn’t need a type parameter. (Case objects with only a type parameter and no value parameters are not allowed in Scala.) This is much closer to the way lists are actually defined in Scala.

(a) Define a list expression that has type List[Any] and would not type-check in the absence of subtyping.
(b) (⋆) Now consider a variant of the list append function append:

```scala
def append1[A](l: List[A], m: List[A]): List[A] = l match {
case Nil => m
case Cons(x,xs) => Cons(x,append1(xs,m))
}
```

Scala’s type inference requires both lists to have the same type, and does not try to cast the two types to a common supertype. Define another function append2 that allows lists of different types as arguments, provided the two element types have a common supertype. (Hint: Scala type parameters can be given type bounds, e.g. def foo[A, B <: A] says that foo has two type parameters, A and B, and B must be a subtype of A.)