Exercises marked ⋆ are more advanced. Please try all unstarred exercises before the tutorial meeting.

1. **Subtyping and Contravariance**
   Consider the following Scala declarations:

   ```scala
   abstract class Shape
   class Rectangle(...) extends Shape
   class Circle(...) extends Shape
   ``

   Thus, `Rectangle <: Shape` and `Circle <: Shape`.

   (a) Suppose we have a function `f: (Shape => Int) => Int`. What could `f` potentially do with its argument? Does the type system allow us to pass a function of type `Rectangle => Int` to `f`?

   (b) Suppose we have a function `g: (Circle => Int) => Int`. What could `g` potentially do with its argument? Does the type system allow us to pass a function of type `Shape => Int` to `g`?

2. **Modules and Interfaces in Scala**
   Consider the following Scala object definition.

   ```scala
   object A {
     type T = Int
     val c: T = 1
     val d: T = 2
     def f(x: T, y: T): T = x + y
   }
   object B {
     type T = String
     val c: T = "abcd"
     val d: T = "1234"
     def f(x: T, y: T) = x + y
   }
   ``

   (a) Write expressions showing how to access each of the elements of `A` and `B`.

   (b) Suppose we execute the import statements

   ```scala
   import A._
   import B._
   ```
after finishing the declaration of $A$. What does unqualified identifier $d$ refer to after that? What if we import in the opposite order?

(c) (+) Construct a Scala trait $ABlike$ defining bindings for all of the components of $A$ and $B$, and so that we can assert that both $A$ and $B$ extend $ABlike$.

(d) (+) Define a function $g$ taking an argument $x$: $ABlike$ that applies $f$ to $c$ and $d$. Apply it to both instances of $Alike$ above. What is its return type?

(e) (+) Create an anonymous instance of $ABlike$ with $T = \text{Boolean}$ and call the function $g$ on it.

3. Type parameters

Some types, such as lists, are naturally thought of as parameterized. For example, in Scala, the type $\text{List}[A]$ takes a parameter $A$, the type of elements of the lists.

Consider the following Scala code:

```scala
abstract class List[A]
case class Nil[A]() extends List[A]
case class Cons[A](head: A, tail: List[A]) extends List[A]
```

This defines a recursive data structure, consisting of lists. (Notice however that $\text{Nil}$ is a case class and so it carries a type annotation and empty parameter list.)

(a) Using the same approach as above, define a type $\text{Tree}[A]$ for binary trees whose leaves are labeled by values of type $A$. There should be two constructors for such trees: $\text{Leaf}(a)$ constructing a leaf with data $a$, and $\text{Node}(t_1, t_2)$ taking two trees and constructing a tree.

(b) Define a recursive function $\text{sum}$ that adds up all of the integers in an $\text{Tree}[\text{Int}]$.

(c) Define a recursive function $\text{map}: \text{Tree}[A] \Rightarrow (A \Rightarrow B) \Rightarrow \text{Tree}[B]$ that applies a given function $f: A \Rightarrow B$ to all of the $A$ values on the leaves of a $\text{Tree}[A]$.

(d) (+) Define a function $\text{flatten}: \text{Tree}[\text{Tree}[A]] \Rightarrow \text{Tree}[A]$.

(e) (+) Define a function $\text{flatMap}: (\text{Tree}[A]) \Rightarrow (A \Rightarrow \text{Tree}[B]) \Rightarrow \text{Tree}[B]$.

4. (+) Ad hoc polymorphism

Traits can also accommodate overloading and reuse of the same name for operations on different types. An operation such as $\text{size}$ can be defined as part of a trait as follows:

```scala
trait HasSize { def size(): Int }
```

(a) Modify the definition of $\text{List}[A]$ above so that it extends $\text{HasSize}$, and define an appropriate size method for it.

(b) Modify the definition of $\text{Tree}[A]$ so that it extends $\text{HasSize}$ and define its size operation.

(c) Write a function $\text{sameSize}$ that takes two values of type $\text{HasSize}$ and checks whether they have the same size.

(d) Call this function on a $\text{List}[\text{Int}]$ and a $\text{Tree}[\text{String}]$ to verify that the correct implementations of $\text{size}$ are called for different types.