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, \(\text{Rectangle} <: \text{Shape}\) and \(\text{Circle} <: \text{Shape}\).

(a) Suppose we have a function \(f: (\text{Shape} \Rightarrow \text{Int}) \Rightarrow \text{Int}\). What could \(f\) potentially do with its argument? Does the type system allow us to pass a function of type \(\text{Rectangle} \Rightarrow \text{Int}\) to \(f\)?

(b) Suppose we have a function \(g: (\text{Circle} \Rightarrow \text{Int}) \Rightarrow \text{Int}\). What could \(g\) potentially do with its argument? Does the type system allow us to pass a function of type \(\text{Shape} \Rightarrow \text{Int}\) 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 ABlike above. What is its return type?

(e) (+) Create an anonymous instance of ABlike with T = 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 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 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 Tree[A] for binary trees whose leaves are labeled by values of type A. There should be two constructors for such trees: Leaf(a) constructing a leaf with data a, and Node(t1, t2) taking two trees and constructing a tree.

(b) Define a recursive function sum that adds up all of the integers in a Tree[Int].

(c) Define a recursive function map: Tree[A] => (A => B) => Tree[B] that applies a given function f: A => B to all of the A values on the leaves of a Tree[A].

(d) (+) Define a function flatten: Tree[Tree[A]] => Tree[A].

(e) (+) Define a function flatMap : (Tree[A]) => (A => Tree[B]) => 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 size can be defined as part of a trait as follows:

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

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

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

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

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