Elements of Programming Languages
Tutorial 4: Subtyping and polymorphism
Week 6 (October 23–27, 2023)

Exercises marked $\star$ are more advanced. Please try all unstarred exercises before the tutorial meeting.

1. **Subtyping and type bounds**

   Consider the following Scala code:

```
abstract class Super
case class Sub1(n: Int) extends Super
case class Sub2(b: Boolean) extends Super
```

   This defines an abstract superclass `Super`, and subclasses with integer and boolean parameters.

   (a) What subtyping relationships hold as a result of the above declarations?

   (b) For each of the following subtyping judgments, write a derivation showing the judgment holds or argue that it doesn't hold.

   i. $Sub1 \times Sub2 <: Super \times Super$
   ii. $Sub1 \to Sub2 <: Super \to Super$
   iii. $Super \to Super <: Sub1 \to Sub2$
   iv. $Super \to Sub1 <: Sub2 \to Super$
   v. ($\star$) $(Sub1 \to Sub1) \to Sub2 <: (Super \to Sub1) \to Super$

   (c) Suppose we have a function

```
def f1(x: Super): Super = x match {
  case Sub1(n) => x
  case Sub2(b) => x
}
```

   that simply inspects the type of the argument but preserves the value. Try running `f1` on `Sub2(true)`. What type does it have? What happens if you try to access the `b` field of the result?

   (d) Now consider a different version of this function:

```
def f2[A](x: A): A = x match {
  case Sub1(n) => x
  case Sub2(b) => x
}
```

   where we have abstracted over the argument type. Does this typecheck? Why or why not? If it typechecks, what happens if we apply it to values of type `Sub1`, `Sub2`, `Int`?

(e) Finally, consider this version:

```
def f3[A <: Super](x: A): A = x match {
  case Sub1(n) => x
  case Sub2(b) =>x
}
```

Here, we have used Scala's support for a feature called *type bounds* to constrain `A` to be a subtype of `Super`, with return type `A`. Does this type-check? Why or why not? If it typechecks, does it solve the problems we encountered with `f1` and `f2`?

2. **Subtyping and Contravariance**

Consider the following Scala declarations:

```
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`?

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:

```
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`, but nodes do not contain `A`-values. There should be two constructors for such trees: `Leaf(a)` constructing a leaf with data a, and `Node(`t_1`,`t_2`)` 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) ($\star$) Define a function `flatten: Tree[Tree[A]] => Tree[A]`.

(e) ($\star$) Define a function `flatMap : (Tree[A]) => (A => Tree[B]) => Tree[B]`