Exercise 1: Merging sorted lists (5 points)

Part 1: Starting recursive

```scala
def merge[T](as: List[T], bs: List[T])(cmp: (T, T) => Boolean): List[T] = (as, bs) match {
  case (Nil, _) => bs
  case (_, Nil) => as
  case (x :: xs, y :: ys) =>
    if (cmp(x, y)) x :: merge(xs, bs)(cmp)
    else y :: merge(as, ys)(cmp)
}
```

Part 2: Going tail-recursive

```scala
def merge2[T](as: List[T], bs: List[T])(cmp: (T, T) => Boolean): List[T] = {
  @tailrec
  def loop(tmpAs: List[T], tmpBs: List[T], tmpRes: List[T]): List[T] = (tmpAs, tmpBs) match {
    case (Nil, _) => tmpRes.reverse ++ tmpBs
    case (_, Nil) => tmpRes.reverse ++ tmpAs
    case (x :: xs, y :: ys) =>
      if (cmp(x, y)) loop(xs, tmpBs, x :: tmpRes)
      else loop(tmpAs, ys, y :: tmpRes)
  }
  loop(as, bs, Nil)
}
```

Exercise 2: Streams (5 points)

Part 1

There are a few possible solutions. Here are 2. One of them uses the function from part 2.

```scala
def iterate[T](x: T)(f: T => T): Stream[T] =
  x #:: iterate(f(x))(f)

def iterate[T](x: T)(f: T => T): Stream[T] =
  iterated(f) map (g => g(x))
```
Part 2

There are a few possible solutions. Here are 2. One of them uses the function from part 1.

```scala
def iterated[T](f: T => T): Stream[T => T] = 
  ((x: T) => x) #:: (iterated(f) map (_ andThen f))
```

```scala
def iterated[T](f: T => T): Stream[T => T] = 
  iterate((x: T) => x)(_ andThen f)
```

Exercise 3: Equational Proof (5 points)

Part 1: axioms for indexWhereAcc

This follows straightforwardly from the implementation:

1. \( \text{indexWhereAcc}(\text{Nil}, f, n) === n \)
2. \( \text{indexWhereAcc}(x :: xr, f, n) === n \) if \( f(x) \) is true
3. \( \text{indexWhereAcc}(x :: xr, f, n) === \text{indexWhereAcc}(xr, f, n+1) \) if \( f(x) \) is false

Part 2: Proof of the lemma

We want to prove:

\[ \text{indexWhereAcc}(xs, f, n) === \text{indexWhereAcc}(xs, f, 0) + n \]

We do it by structural induction on \( xs \).

**Nil case:**

\[ \text{indexWhereAcc}(\text{Nil}, f, n) =?= \text{indexWhereAcc}(\text{Nil}, f, 0) + n \]

\[ || (1) || (1) \]

\[ n =?= 0 + n \]

**x :: xr case with \( f(x) \) is true:**

\[ \text{indexWhereAcc}(x :: xr, f, n) =?= \text{indexWhereAcc}(x :: xr, f, 0) + n \]

\[ || (2) || (2) \]

\[ n =?= 0 + n \]

**x :: xr case with \( f(x) \) is false:**

\[ \text{indexWhereAcc}(x :: xr, f, n) =?= \text{indexWhereAcc}(x :: xr, f, 0) + n \]

\[ || (3) || (3) \]

\[ \text{indexWhereAcc}(xr, f, n+1) =?= \text{indexWhereAcc}(xr, f, 1) + n \]

\[ || \text{inductive hypot.} || \text{inductive hypot.} \]

\[ \text{indexWhereAcc}(xr, f, 0) + (n+1) =?= (\text{indexWhereAcc}(xr, f, 0) + 1) + n \]

\[ || \text{arithmetics} || \text{arithmetics} \]

\[ \text{indexWhereAcc}(xr, f, 0) + n + 1 =?= \text{indexWhereAcc}(xr, f, 0) + 1 + n \]
Part 3: Proof that the implementation satisfies the spec

Nil case:

\[
\text{indexWhere}(\text{Nil}, f) = 0 \\
\text{indexWhereAcc}(\text{Nil}, f, 0) = 0 \\
0 = 0
\]

\[\text{x :: xr case with } f(x) \text{ is true:}\]

\[
\text{indexWhere}(x :: xr, f) = 0 \\
\text{indexWhereAcc}(x :: xr, f, 0) = 0 \\
0 = 0
\]

\[\text{x :: xr case with } f(x) \text{ is false:}\]

\[
\text{indexWhere}(x :: xr, f) = 1 + \text{indexWhere}(xr, f) \\
\text{indexWhereAcc}(x :: xr, f, 0) = 1 + \text{indexWhereAcc}(xr, f, 0) \\
\text{indexWhereAcc}(xr, f, 1) = 1 + \text{indexWhereAcc}(xr, f, 0) \\
\text{indexWhereAcc}(xr, f, 0) + 1 = 1 + \text{indexWhereAcc}(xr, f, 0)
\]

Exercise 4: Subtyping (5 points)

Union of Sets

\[
def \text{union[A1 >: A]}(other: \text{Set[A1]}): \text{Set[A1]}
\]

Explanation: For a detailed explanation, see lecture on covariance.

- If we set other to \text{Set[A]}, the expression \text{val fruits = Set(new Apple).union(Set(new Peach))} will not typecheck.
- We therefore need a new type A1 to authorize any type for the elements of other.
- However, we also want to ensure that the elements of this can be in a \text{Set[A1]}. Hence the constraint that A1 >: A.

Function Conformance

Explanation: for an assignment \text{val x: T = bla} to be valid, the type of bla must be a subtype of T. In all the exercises below, we need to verify this.

We also have the subtyping relation for functions:

We also know that the function type notation is right associative, i.e. A => B => C is the same as A => (B => C)

1. Is A => D <: B => D? yes, because of the above
2. Is A => (D => C) <: A => (C => D)?
   • A >: A.
   • So is D => C <: C => D? No, because D <: C
3. Is (D => A) => B <: (D => B) => A?
   • Is (D => A) >: (D => B)? Yes, because D <: D and A >: B.
   • Is B <: A? Yes.

Exercise 5: Flattening (5 points)

The important part in this exercise was to pattern match an element of ls correctly. The naive solution is given below. It is quadratic in the number of “leaves” in ls.

def flatten(ls: List[Any]): List[Int] = ls match {
  case Nil => Nil
  case (x: Int) :: xs => x :: flatten(xs)
  case (x: List[Any]) :: xs => flatten(x) ++ flatten(xs)
}

There is a solution that is linear in the number of “leaves”:

def flatten(ls: List[Any]): List[Int] = {
  def flattenConcat(tmpList: List[Any], tail: List[Int]): List[Int] = tmpList match {
    case Nil => tail
    case (x: Int) :: xs => x :: flattenConcat(xs, tail)
    case (x: List[Any]) :: xs => flattenConcat(x, flattenConcat(xs, tail))
  }
  flattenConcat(ls, Nil)
}

A MatchError is thrown if the pattern match fails on a certain pattern, there is no need to explicitly add a default case.