# Exercise sheet 08— Solutions

## 1 Tuple composition for Applicative

As both applicatives are in a tuple and not nested, we can handle them completely independently from each other.

In `pure` we simply wrap the passed A once in F and once in G, and return the results in a tuple.

```
def pure[A](x: A): (F[A], G[A]) =
  (Applicative[F].pure(x), Applicative[G].pure(x))
```

In `ap` we already get two tuples, we apply `F.ap` to the first tuple elements and `G.ap` to the second elements.

```
def ap[A, B](ffgg: (F[A => B], G[A => B]))(faga: (F[A], G[A])): (F[B], G[B]) =
    (ffgg._1 <*> faga._1, ffgg._2 <*> faga._2)
```

Here we use the operator syntax provided by Cats:
`ffgg._1 <*> faga._1` is equivalent to `Applicative[F].ap(ffgg._1, faga._1)`.

If we use our `ap` implementation based on `map2` from the lecture, we can also implement `map2`, here we also use the implementations of the respective F and G applicatives:

```
override def map2[A,B,C](faga: (F[A], G[A]), fbgb: (F[B], G[B]))(f: (A, B) => C) =
  (Applicative[F].map2(faga._1, fbgb._1)(f),
   Applicative[G].map2(faga._2, fbgb._2)(f))
```

or with additional Cats syntax:

```
override def map2[A,B,C](faga: (F[A], G[A]), fbgb: (F[B], G[B]))(f: (A, B) => C) =
  ((faga._1, fbgb._1).mapN(f),
   (faga._2, fbgb._2).mapN(f))
```

# 2 Applicative Combinators

As a reminder, the signature of product is:

```
def product[F[_],A,B](fa: F[A], fb: F[B])(using Applicative[F]): F[(A,B)]
```

### 2.1 ap via product and map

```
def apViaProductAndMap[F[_],A,B](ff: F[A => B])(fa: F[A])(using Applicative[F]): F[B] =
  Applicative[F].product(ff, fa).map((fab, a) => fab(a))
```

First we combine both `F`s with `product` and get a `F[(A => B, A)]`. On this single `F` we can now use `map` to apply the function to the value.

### 2.2 product via ap

First we define `ap` via `product` and `map`:

```
def productViaApAndMap[F[_],A,B](fa: F[A], fb: F[B])(using Applicative[F]): F[(A,B)] =
  fa.map((a: A) => (b: B) => (a, b)) <*> fb
```

With `map` we transform the `A` into a function taking a `B` and putting it into a tuple together with the A. We can then apply that function to the `F[B]` with `<*>`, i.e. `ap`.

To have it only use `ap` but not `map`, we replace the latter with the `map` definition via `ap` from the lecture:

```
def productViaApAndPure[F[_],A,B](fa: F[A], fb: F[B])(using Applicative[F]): F[(A,B)] =
  Applicative[F].pure((a: A) => (b: B) => (a, b)) <*> fa <*> fb
  //                F[A => B => (A, B)]             F[A]   F[B]    -> F[(A, B)]
```

# 3 Applicative instance for binary trees

### 3.1 Applicative for binary trees

First, the `map` method from the Functor exercise sheet:

```
override def map[A, B](fa: Tree[A])(f: A => B): Tree[B] =
  fa match
    case Branch(left, right) => Branch(left map f, right map f)
    case Leaf(value) => Leaf(f(value))
```

We now want to implement `pure` and `ap`. In `pure` we put the given value into a `Leaf`:

```
override def pure[A](x: A): Tree[A] = Leaf(x)
```

In `ap` we now have two trees we have to traverse. As we already have `map`, which takes a function `A => B`, it lends itself to traverse the tree with such functions first and apply them to the other tree via `map`:

```
override def ap[A, B](ff: Tree[A => B])(fa: Tree[A]): Tree[B] =
  ff match
    case Branch(left, right) => Branch(left <*> fa, right <*> fa)
    case Leaf(f) => fa.map(f)
```

So we match on `ff` and call `ap` recursively with the tree `fa` in case of a branch. When we arrive at a leaf, we take the function stored in it and apply it to our `Tree[A]` using `map`.

### 3.2 map2 on binary trees

To understand what `map2` does for binary trees, we first look at `map2` on an easier data structure, namely List.

Let's compare the List methods `product` (= `map2` with fixed `f`) and `zip`:

```
def product[A,B](fa: List[A], fb: List[B]): List[(A,B)]
def zip[A,B](fa: List[A], fb: List[B]): List[(A,B)]
```

The signatures are identical, but the two methods have different results. `zip` combines every element from the first list with exactly one element from the other list.

```
val fa = List(1, 2, 3)
val fb = List('a', 'b', 'c', 'd', 'e')
fa.zip(fb) == List((1, 'a'), (2, 'b'), (3, 'c'))
```

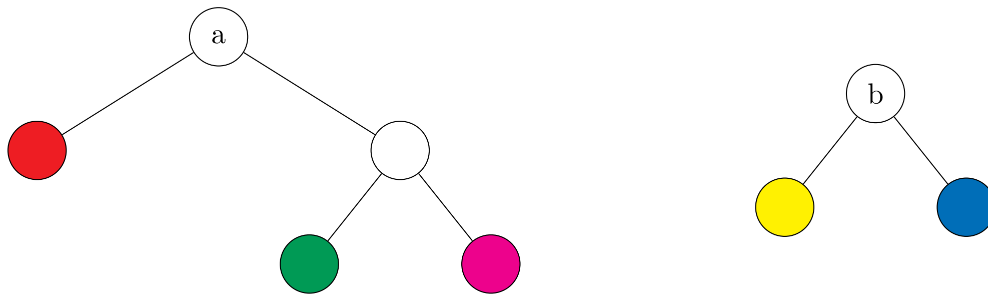
`product` on the other hand combines *every* elemenent of the first list with *every* element of the second list. We get something like the lists' "cross product":

```
Applicative[List].product(fa,fb) == List(
    (1, 'a'), (1, 'b'), (1, 'c'), (1, 'd'), (1, 'e'),
    (2, 'a'), (2, 'b'), (2, 'c'), (2, 'd'), (2, 'e'),
    (3, 'a'), (3, 'b'), (3, 'c'), (3, 'd'), (3, 'e')
  )
```

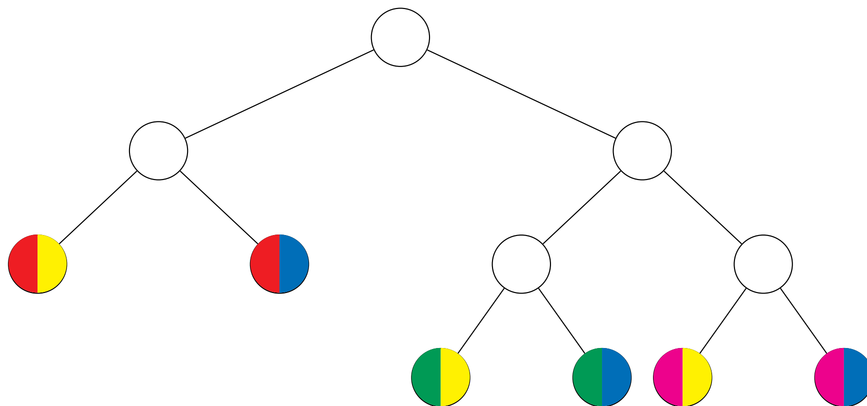
It's similar for our trees. If we combine them with `map2`, we get the tree structure, that results from replacing every leaf in the structure of the first tree with the full structure of the second tree.

(See next page for an illustrated example)

An example for map2 with two trees $a$ and $b$:




When we call `a.map2(b)(_ |+| _)` on our trees (where `|+|` is an arbitrary operation combining the two elements, displayed here as the combination of node colors) we get a tree, which begins at its root with the same structure as $a$. At each location, where $a$ has a leaf, we now have a branch with two leaves: $b$'s structure:



If we swap the trees, calling `b.map2(a)(_ |+| _)`, we get a different result. So our `map2` is not commutative: