Exercise sheet for lecture 06— Foldable and Functor

In this exercise we look at typeclasses, givens and cats [1]. You can find some of the signatures for the exercises as well as the given implementations online at https://gitlab2.informatik.uni-wuerzburg.de/intro-to-fp/tasksheets as a Git repository.

1 Typeclass Instances for Binary Trees

In the file `Tree.scala` you can find an implementation of a binary tree (a simplified version of the trees you may know from the *Huffman* exercise). We will implement some instances for `Tree`, using the typeclass traits from Cats that are equivalent to the ones known from the lecture. Cats is one of the most popular Scala libraries for functional programming. Also note the API docs at .

Let's start with the definition of our binary trees:

```
enum Tree[+A]:
  case Branch[A](left: Tree[A], right: Tree[A])
  case Leaf[A](value: A)
```

If Cats is included in a project, you can find the typeclasses provided by it under the package `cats`, in particular the ones known from the lecture. In the package `cats.syntax` there are additional objects which enable us to use the typeclasses' methods in the familiar object oriented syntax (e.g. `l.map(_ * 2)` instead of `Functor[List].map(l)(_ * 2)`)[2]. In the Git repository the required imports are already included.

IntelliJ is sometimes unable to cope with the definitions from `cats.syntax` and marks formally correct code as wrong. Sadly the only way around that is to verify errors shown by IntelliJ with the compiler (e.g. via `sbt compile`) and ignore the wrong ones, or using an editor that supports the Metals Language Server.

a) In the lecture we saw monoids and the typeclass `MonoidK` for monoids on type constructors. Think about why our binary trees aren't monoids!

   A more general variant of monoids are semigroups, which are very similar but don't define a zero element. So they consist only of a set and an associative operation on that set. There is a matching `Semigroup` typeclass in Cats. Like with monoids, there also is a `SemigroupK` variant for higher kinded types.

   Implement an instance of `SemigroupK` for `Tree`. Note that `combineK` has to be associative!

b) With `Functor`, Cats has a typeclass for covariant functors, like we know from the lecture. It works like the one known from the lecture, except for `map` not being an extension method, but taking the datastructure as a parameter instead. Thanks to the `syntax` package, the method can still be used like if it was an extension method.

   Implement an instance of `Functor` for `Tree`.

c) Cats also has a same-named equivalent of the `Foldable` typeclass, which works similarly.

[1] `cats` is a library for functional programming
[2] As Cats is still compatible with Scala 2, those sadly aren't defined as extensions, so the extra import is needed

In `Foldable`'s `foldRight`, Cats uses the `Eval` class for stacksafe non-strict evaluation. `Eval` is not part of the lecture and mostly irrelevant for this exercise too. If you are interested in how it works, have a look at the documentation. For your implementation, you can ignore `Eval`. At no point you will need to create `Eval` instances and you can treat `Eval[B]` just like `B` in foldRight..

Implement an instance of `Foldable` for `Tree`. *Hint:* in this case `foldLeft` needn't be tailrecursive.

2 A semiautomatic monoid factory

Please note: The following definition uses mathematical notation for monoids: a tuple consisting of a set (in our case a type), the operation and the zero element.

A monoid isomorphism is a bijection $f : A \to B$ between two monads $(A, \oplus, z_A)$ and $(B, \odot, z_B)$ with the following properties:

- $\forall x, y \in A : f(x \oplus y) = f(x) \odot f(y)$,
- $f(z_A) = z_B$.

We can use these properties, to use a given monoid isomorphism to create new monoid instances.

Implement the function `imap`, which takes a function `f: A => B`, its inverse `g` and a monoid for `A`, and creates a monoid for `B` from those. Note that in Cats the zero element is called `empty`.

```
def imap[A, B](f: A => B, g: B => A)(using Monoid[A]): Monoid[B] = ???
```

To test your implementation, you are given a class `Box[A]`, which “wraps” arbitrary values of type `A`:

```
case class Box[+A](value: A)
```

Implement a monoid instance for `Box[A]` using `imap`, where `A` has a monoid.

```
given boxMonoid[A](using Monoid[A]): Monoid[Box[A]] = ???
```

Hint: You can find the instructions and a `main` method in the file `InvariantMonoid.scala` in Git.