Exercise sheet for lecture 05— Algebras and the Monoid Typeclass

The topics of this exercise are algebras, their laws, and the type class `Monoid`. Die Signaturen der in den Aufgaben geforderten Methoden sowie die vorgegebenen Implementierungen finden Sie online unter `https://gitlab2.informatik.uni-wuerzburg.de/intro-to-fp/tasksheets` als Git-Repository.

1 Monoid Instance for Functions

Implement an instance of `Monoid` for functions, using the following signature:

```
given functionMonoid[A,B](using B: Monoid[B]): Monoid[A => B]
```

Consider the required return type for `combine` and `zero` and how you can produce it.

"‘Follow the types!"’

2 Word Count — Parallel parsing

This exercise is a bit more difficult or extensive, but a more concrete programming task, based on section 10.4 from the book "‘Fuctional Programming in Scala"’ by Paul Chiusano and Rúnar Bjarnason.

Task description:

Say we want to count the number of words in a `String`, a simple parsing task. One could iterate though the String character by character, looking for whitespace and counting the number of unbroken substrings there are. With such sequential parsing, the parser state would only need to remember if the previously seen character was a whitespace.

Suppose we don't want to do it just for a short string, but for a huge text file, maybe even too large to fit into memory. In such a case, it would be useful to work with smaller parts of the file in parallel. The idea would be to divide the file into *chunks*, handle several of those chunks in parallel and combine the results. In that case, the parser's state needs to be a bit more complex, and we need a way to combine intermediate results, regardless of wether the current chunk is at the start, the middle or the end of the file. That means, our combining step should be *associative*.

Example: We look at a short sentence and imagine it being a large file:

`"lorem ipsum dolor sit amet, "`

If we split a string about in the middle, it's possible that we split a word apart. In this examle we'd get `"lorem ipsum do"` and `"lor sit amet, "`. In the combination step, we'd like to prevent the word `dolor` to be counted twice. Simply counting the words as `Int` seems to not be enough. So we need a data structure, that can represents partial results with possibly divided words like `do` and `lor`, and can remember the number of already seen full words like `ipsum`, `sit` and `amet`.

We could represent partial results of the word count as the following algebraic data type:

```
enum WordCount:
  case Stub(chars: String)
  case Part(lStub: String, words: Int, rStub: String)
```

A `Stub` is the most simple case, containing a possibly incomplete word. Such a `Stub` will never contain whitespace, only real characters / partial words. A `Part` on the other hand stores the number of seen complete words in `words`. The value `lStub` contains a partial word seen left of the counted whole words, `rStub` one on the right of those words.

For example, if we count words in the string `"lorem ipsum do"`, the result would be `Part("lorem", 1, "do")`, because we only have one word that is definitely complete. As there is no whitespace left of `lorem` or right of `do`, we can't be sure if they are complete words (remember, we don't know if the string is the start or end of the file), so they haven't been counted yet. With `"lor sit amet, "` (note the whitespace in the end) we'd get the result `Part("lor", 2, "")`.

a)

Implement a Monoid instance for WordCount and make sure it fulfills the *monoid laws*. We recomment to use pattern matching in `combine`.

```
given Monoid[WordCount] with
    def zero = ???
    def combine(a: WordCount, b: WordCount) = ???
```

Think about, how to combine two `Part` objects, so that it results in the desired behaviour of counting whole words, especially what happens with the `rPart` of the left Part and the `lPart` of the right Part.

b)

Use the monoid to implement a function, which counts words in a string by dividing it recursively into substrings (of approximately same size) and counting the words in those substrings.

```
def count(str: String)(using WM: Monoid[WordCount]): Int
```

Hints:

- Here, using several internal helper functions in `count` can be useful. For example a function, that takes a single `Char` and returns a `WordCount` object. `def wordCount(c: Char): WordCount = ???`
- To check if a character is a whitespace, you can use the `.isWhitespace` method.
- A string can be divided into substrings at a given position with `.splitAt` (returns a tuple of two strings).
- In the next lecture, we'll see a variant of `fold` specially for use with monoids, which handles the recursion needed here, but in this exercise you'll still have to do it manually.