Übungsblatt zu Vorlesung 05— Algebras and the Monoid Typeclass

In dieser Übung beschäftigen wir uns mit Algebren, ihren Gesetzen und der Typklasse `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-Instanz für Funktionen

Implementieren Sie eine Instanz von `Monoid` für Funktionen mit der folgenden Signatur.

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

Überlegen Sie sich, welchen Rückgabetyp `combine` und `zero` haben müssen und wie dieser erzeugt wird.

„Follow the types!“

2 Wörter zählen — paralleles Parsen

Diese Aufgabe ist wieder etwas schwerer bzw. umfangreicher, aber dafür eine konkrete Programmieraufgabe angelehnt an Abschnitt 10.4 im Buch „Fuctional Programming in Scala“ von Paul Chiusano und Rúnar Bjarnason.

Aufgabenstellung:

Angenommen, man wollte die Anzahl an Wörtern in einem `String` zählen, eine einfach Parsing-Aufgabe. Man könnte den String Zeichen für Zeichen durchgehen, nach Leerzeichen suchen und zählen, wie viele ununterbrochene Zeichenketten es gibt. Bei solch sequenziellem Parsing könnte der *parser* State sich einfach merken ob der letzte gesehene character ein Leerzeichen war.

Angenommen, man würde das aber nicht nur für einen kurzen String anwenden sondern eine riesige Textdatei, gegebenenfalls zu groß um in den Speicher zu passen. Dafür wäre es sinnvoll, mit Teilen der Datei parallel zu arbeiten. Die Idee wäre demnach, die Datei in *chunks* zu unterteilen, mehrere dieser chunks parallel zu verarbeiten und die Ergebnisse zu kombinieren. In diesem Fall müsste der Zustand des Parsers etwas komplexer sein und man müsste Zwischenergebnisse kombinieren können unabhängig davon ob der aktuelle Abschnitt am Beginn, in der Mitte oder am Ende der Datei ist. Das heißt wir wollen, dass der kombinierende Schritt *assoziativ* ist.

Beispiel Wir betrachten einen kurzen Satz und stellen uns vor es wäre eine große Datei:

`"lorem ipsum dolor sit amet, "`

Wenn ein String ungefähr in der Hälfte unterteilt wird, kann es passieren, dass wir mitten in einem Wort teilen. In diesem Beispiel bekämen wir also `"lorem ipsum do"` und `"lor sit amet, "`. Im Kombinationsschritt wollen wir vermeiden, dass das Wort `dolor` zweimal gezählt wird. Die Wörter einfach nur als `Int` zu zählen scheint also nicht zielführend zu sein. Das heißt wir benötigen eine Datenstruktur, die mit Teilergebnissen wie den halben Wörtern `do` und `lor` umgehen kann und sich die Anzahl bisher gesehener ganzer Wörter merken kann, wie z.B. `ipsum`, `sit` und `amet`.

Das Teilergebnis des Word Counts könnte z.B. durch einen Algebraischen Datentyp dargestellt werden:

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

Ein `Stub` ist der einfachste Fall, der ein möglicherweise durch Auftrennung unvollständiges Wort enthält. Ein `Stub` enthällt nie ein Leerzeichen, sondern immer nur echte Buchstaben/Teilwörter. Ein `Part` dagegen speichert die Anzahl bisher gesehener, vollständiger Wörter in `words`. Der Wert `lStub` hält ein Teilwort, das links von diesen Wörtern gesehen wurde und `rStub` ein Teilwort rechts von diesen Wörtern.

Würden wir beispielsweise auf dem String `"lorem ipsum do"` zählen, wäre das Ergebnis `Part("lorem", 1, "do")`, denn es gibt nur ein Wort, das mit Sicherheit vollständig ist. Da kein Leerzeichen links von `lorem` oder rechts von `do` ist, können wir nicht sichergehen, ob es vollständige Wörter sind (wir wissen nicht, ob der String am Anfang oder Ende der Datei ist) , weshalb sie noch nicht gezählt wurden. Mit `"lor sit amet, "` (Leerzeichen am Ende beachten) bekämen wir das Ergebnis `Part("lor", 2, )`.

a)

Implementieren Sie eine Monoid Instanz für WordCount und gehen Sie sicher, dass sie den *Monoid Laws* genügt. Arbeiten Sie in `combine` am besten mit Pattern Matching.

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

Überlegen Sie sich, wie die Kombination zweier `Part`-Objekte aussehen muss, um das gewünschte Verhalten des Wörter Zählens zu erhalten, insbesondere, was mit dem `rPart` des linken Parts und dem `lPart` des rechten Parts passiert.

b)

Benutzen Sie den Monoid, um eine Funktion zu implementieren, die die Wörter in einem String zählt indem sie den String rekursiv in (etwa gleich große) Substrings unterteilt und die Wörter in diesen Substrings zählt.

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

Hinweise:

- Hier können mehrere interne Funktionen innerhalb von `count` sinnvoll sein. Beispielsweise eine grundlegende Funktion, die einen einzelnen `Char` liest und ein `WordCount` Objekt zurückgibt. `def wordCount(c: Char): WordCount = ???`
- Mit `.isWhitespace` können Sie überprüfen, ob ein Char ein Leerzeichen ist.
- Mit `.splitAt` kann man einen String an einer gegebenen Stelle aufteilen (gibt ein Tupel von zwei Strings zurück).

- In der nächsten Vorlesung werden wir eine Variante von `fold` speziell für Verwendung mit Monoiden sehen, welche die hier nötige Rekursion abdeckt, in diesem Blatt muss sie noch manuell implementiert werden.