Übungsblatt 05— Lösungen

1 Monoid-Instanz für Funktionen

```scala
given functionMonoid[A, B](using B: Monoid[B]): Monoid[A => B] with
    def zero: A => B = _ => B.zero

    def combine(f: A => B, g: A => B): A => B = a => f(a) |+| g(a)
```

Die Idee für `zero` ist einfach. Wir brauchen eine Funktion, die ein `B` erzeugt. So eine Funktion können wir bauen, weil wir einen `Monoid[B]` zur Verfügung haben. Damit können wir eine Funktion nehmen, die den Input einfach ignoriert und das zero-B zurück gibt. Interessanterweise gibt es auch keine andere Möglichkeit, als den Input zu ignorieren. Der Input ist vom generischen Typ `A` und wir können nichts mit einem `A` machen, wenn wir nicht mehr darüber wissen. *Parametricity strikes again!*

Für `combine` kann man wie folgt vorgehen: Wir wissen, dass wir zwei Funktionen haben und beide Funktionen benutzen müssen. Warum wissen wir das? Wenn wir nur die erste Funktion benutzen, dann ergäbe `combine(zero, g)` sicher nicht `g` (weil wir die nicht benutzen), obwohl es das müsste, weil der andere Parameter das Zero-Element ist. Analog für den Fall, dass wir nur die zweite Funktion benutzen. Wenn wir beide Funktionen benutzen wollen, müssen wir ihnen ein `A` geben. Wir haben aber nur ein `A`, nämlich den Input. Geben wir den an beide Funktionen haben wir zwei `B`. Jetzt müssen wir diese noch vereinen, weil wir beide verwenden müssen. Die einzige Möglichkeit das zu tun ist `Monoid[B].combine` zu benutzen.

2 WordCount

a)

Zunächst widmen wir uns dem Monoid für `WordCount`:

```scala
given Monoid[WordCount] with
  val zero = Stub("")

  def combine(a: WordCount, b: WordCount) = (a, b) match
    case (Stub(c), Stub(d)) => Stub(c + d)
    case (Stub(c), Part(l, w, r)) => Part(c + l, w, r)
    case (Part(l, w, r), Stub(c)) => Part(l, w, r + c)
    case (Part(l1, w1, r1), Part(l2, w2, r2)) =>
      Part(l1, w1 + (if ((r1 + l2).isEmpty) 0 else 1) + w2, r2)
```

Wenn wir zwei `Stub`s haben, werden diese einfach zu einem längeren `Stub` zusammengehängt. Hierbei muss man sich daran erinnern, dass ein `Stub` nie Leerzeichen enthält. Wenn man zwei von ihnen bekommt, kann man sie also einfach zu einem neuen `Stub` vereinen, da so nie Wortgrenzen verloren gehen können. Erhält man einen `Stub` und einen `Part`, kann der `Stub` aus dem gleichen Grund einfach angehängt werden. Erhält man zwei `Part`s, so können diese vereint werden. Hierbei wird der linke Rand des ersten `Part`s behalten und der rechte Rand des zweiten. Die Zahl der gefundenen Wörter wird addiert. Sind die beiden Ränder in der Mitte leer, so bleibt es dabei. In diesem Fall nähern sich die beiden `Part`s jeweils mit Leerzeichen aneinander an. Es ergibt sich also kein neues zusätzliches Wort. Ist das nicht der Fall, so ergibt sich aus den beiden Rändern in der Mitte ein neues Wort, und der WordCount wird um eins erhöht.

Das Identity-Element ist der leere Stub. Das ist leicht zu sehen: Nachdem jede Operation mit einem Stub lediglich den String des Stubs an etwas anderes anhängt und den leeren String irgendwo anzuhängen aber nie etwas ändern, lässt eine Operation mit dem leeren Stub immer den anderen Operanden unverändert zurück.

b)

Widmen wir uns nun der `count`-Funktion:

```
def count(str: String)(using WM: Monoid[WordCount]): Int =
  def wordCount(c: Char): WordCount =
    if (c.isWhitespace)
      Part("", 0, "")
    else
      Stub(c.toString)

  def recSplit(part: String): WordCount =
    if part.length > 1 then
      val (left, right) = part.splitAt(part.length / 2)
      recSplit(left) |+| recSplit(right)
    else if part.length == 0 then
      WM.zero
    else
      wordCount(part.head)

  def unstub(s: String): Int = s.length.min(1)

  recSplit(str) match
    case Stub(s) => unstub(s)
    case Part(l, w, r) => unstub(l) + w + unstub(r)
```

Der Aufbau der Funktion lässt sich wie folgt beschreiben: Wir teilen den Eingangsstring so lange rekursiv in zwei Hälften auf, bis wir einzelne `Char`s haben und wandeln jeden `Char` zu einem `WordCount`. Handelt es sich dabei um einen normalen Buchstaben, so wird ein `Stub` erzeugt. Handelt es sich um ein Leerzeichen, so wird ein leerer `Part` erzeugt.

Die Ergebnisse der rekursiven Aufrufe vereinen wir mit dem zuvor definiert Monoiden.

Am Ende schauen wir uns den verbleibenden `WordCount` an. Handelt es sich dabei um einen `Stub`, so haben wir entweder gar kein Wort, falls der String leer war, oder wir haben genau ein Wort, in allen anderen Fällen. Da ein `Stub` keine Leerzeichen enthalten kann, gibt es keine weiteren Möglichkeiten.

Erhalten wird am Ende einen `Part`, so haben wir auf jeden Fall die Anzahl an Wörtern zwischen den beiden Rändern des `Part`. Hinzu kommt ein weiteres Wort, falls der linke Teil des `Part`s nicht leer ist, und ein weiteres, falls der rechte Teil nicht leer ist.