Functional Reduction

Was sind Monoide in Scala?

= algebraische Struktur mit

- einer assoziativen Verknüpfung
- einem neutralen Element

```
trait Monoid[M] {
  val zero: M // neutrales Element
  def op(a: M, b: M): M // assoziative Verknüpfung
}
```

```
object Monoid {
  def apply[M](z: M, operator: (M, M) => M): Monoid[M] =
    new Monoid[M] {
      override def op(a: M, b: M): M = operator.apply(a, b)
      override val zero: M = z
    }
```

Beispiele:

```
val intPlusMonoid: Monoid[Int] = Monoid(0, (a, b) => a + b)
val intTimesMonoid: Monoid[Int] = Monoid(1, (a, b) => a * b)
val doublePlusMonoid: Monoid[Double] = Monoid(0.0, (a, b) => a + b)
```

Was macht ReduceLeft und ReduceRight?

- `reduceLeft` und `reduceRight` sind Funktionen, die eine Liste von Elementen auf ein einzelnes Element reduzieren

Rekursiv:

```
def reduceRight[A](as: List[A])(monoid: Monoid[A]): A =
    as match {
        case Nil => monoid.zero
        case head :: tail => monoid.op(head, reduceRight(tail)(monoid))
    }
```

```
import Monoid.*
val lst = List(1, 2, 3, 4, 5)
val sum = reduceRight(lst)(intPlusMonoid) // 15
val str = reduceRight(lst.map(_.toString))(Monoid("", (a, b) => a + b)) //
"12345"
```

Iterativ:

```
def reduceLeft[A](as: List[A])(monoid: Monoid[A]): A = {
    var result = monoid.zero
    for (a <- as) {
        result = monoid.op(result, a)
    }
    result
}
```

```
import Monoid.*
val lst = List(1, 2, 3, 4, 5)
val sum = reduceLeft(lst)(intPlusMonoid) // 15
val str = reduceLeft(lst.map(_.toString))(Monoid("", (a, b) => a + b)) //
"12345"
```

Was sind Kontext Parameter oder given in Scala?

Kontext Parameter erlauben es Werte aus dem Kontext zu verwenden, ohne sie explizit zu übergeben. Sie sind eine Erweiterung von impliziten Parametern und können in Kombination mit impliziten Parametern verwendet werden.

- `using`: fordert eine implizite Instanz
- `given`: definiert eine implizite Instanz

```
trait Monoid[A] {
  def zero: A
  def combine(x: A, y: A): A
}

// Definiere eine `given`-Instanz für `Int`
given intAdditionMonoid: Monoid[Int] with {
  def zero: Int = 0
  def combine(x: Int, y: Int): Int = x + y
}

// Methode mit Kontext-Parameter
def sumList[A](list: List[A])(using monoid: Monoid[A]): A =
  list.foldLeft(monoid.zero)(monoid.combine)
```

```
// Aufruf der Funktion (ohne explizites monoid!)
@main def run() = {
  val numbers = List(1, 2, 3, 4)
  println(sumList(numbers)) // Automatisch intAdditionMonoid → Ausgabe: 10
}
```

Was versteht man unter Lifting von Monoiden?

= bestehende Monoid-Instnaz auf eine komplexere Struktur anzuheben, zb durch Verpacken in Option, List, etc.

```
trait Monoid[A] {
  def zero: A
  def combine(x: A, y: A): A
}

given intAdditionMonoid: Monoid[Int] with {
  def zero: Int = 0
  def combine(x: Int, y: Int): Int = x + y
}

given optionMonoid[A](using monoid: Monoid[A]): Monoid[Option[A]] with {
  def zero: Option[A] = None
  def combine(x: Option[A], y: Option[A]): Option[A] =
    (x, y) match {
      case (Some(a), Some(b)) => Some(monoid.combine(a, b)) // Nutze
ursprüngliches Monoid
      case (Some(a), None)    => Some(a)
      case (None, Some(b))    => Some(b)
      case (None, None)       => None
    }
}
```

```
def sumOptions[A](list: List[Option[A]])(using monoid: Monoid[Option[A]]):
Option[A] =
  list.foldLeft(monoid.zero)(monoid.combine)

@main def run() = {
  val numbers: List[Option[Int]] = List(Some(1), None, Some(3), Some(4))
  println(sumOptions(numbers)) // Ausgabe: Some(8)
}
```

-> Monoid für Int wurde gehoben, sodass es auch mit Option[Int] funktioniert, funktioniert mit beliebigen Typen, wenn eine given Instanz für den Typen existiert

Wie setzt man Parallele Reduktion in Scala um?

//TODO