Exercise 1: Graph coloring (10 points)

```
def var(id: Int, color: String) = Var(s"$id$color")

def graphColoring(graph: Graph): Formula = {

  val allConstraints = for {
    Vertex(id, neighbors) <- graph.vertices
  } yield {

    val oneColor: Formula = Or(var(id, "R"), Or(var(id, "G"), var(id, "B")))

    val onlyOneColor: Formula = And(
      Or(
        Not(var(id, "R")),
        Not(var(id, "G"))),
      And(
        Or(
          Not(var(id, "R")),
          Not(var(id, "B"))),
        Or(
          Not(var(id, "G")),
          Not(var(id, "B")))))

    val noDuplicateNeighbors = for {
      n <- neighbors
      if n <= id
    } yield {
      And(
        Not(And(var(id, "R"), var(n, "R"))),
        And(
          Not(And(var(id, "G"), var(n, "G"))),
          Not(And(var(id, "B"), var(n, "B"))))
    }

    noDuplicateNeibors.foldLeft[Formula](And(oneColor, onlyOneColor))(And(_, _))
  }

  val True = Or(Var("2B"), Not(Var("2B")))
  allConstraints.foldLeft[Formula](True)(And(_, _))
}
```

Exercise 2: Streams (10 points)

```
def pairAverages(data: Stream[Double]): Stream[Double] = data.take(2) match {
  case Seq(a, b) => ((a + b) / 2.0) #:: pairAverages(data.tail)
  case _ => Stream.empty[Double]
}

def windowAverage(windowSize: Double, data: Stream[Double]): Stream[Double] = {
  val n = windowSize.toInt
  val init = data.take(n)
  if (init.size == n) {
    (init.sum / n) #:: windowAverage(windowSize, data.tail)
  }
  else {
    Stream.empty[Double]
  }
}

def rollingAverage(data: Stream[Double]): Stream[Double] = {

  def sumAndCount(sumAcc: Double, countAcc: Double, stream: Stream[Double]):
      Stream[(Double, Double)] = stream.headOption match {

    case None => Stream.empty[(Double, Double)]
    case Some(x) => {
      val newSumAcc = sumAcc + x
      val newCountAcc = countAcc + 1

      (newSumAcc, newCountAcc) #:: sumAndCount(newSumAcc, newCountAcc, stream.tail)
    }
  }

  sumAndCount(0, 0, data).map { case (sum, count) => sum / count }
}
```

Exercise 3: Variable Substitution in Lisp (10 points)

```
def substitute(term: Any, symbol: Symbol, replaceBy: Any): Any = term match {
  case s: Symbol if s == symbol => replaceBy
  case xs: List[Any] => xs.map(substitute(_, symbol, replaceBy))
  case _ => term
}
```

```
(def (substitute term symbol replaceBy)
  (cond
    ((= term symbol) replaceBy)
    ((isCons? term) (cons
      (substitute (car term) symbol replaceBy)
      (substitute (cdr term) symbol replaceBy)))
    (else term))
rest)
```