Exercise 1: Lambda Calculus

Exercise 1.1

Answer: (evaluates to 3)

```
(def (toInt n)
  (n (lambda (x) (+ 1 x)) 0)

(val three (lambda (f x) (f (f (f x))))
(toInt three)
))
```

Exercise 1.2

Implement multiplication on church numerals.

Answer: (evaluates to 6)

```
(def (mult m n)
  (lambda (f x)
    (m (lambda (y) (n f y)) x)
  )

(def (toInt n)
  (n (lambda (x) (+ 1 x)) 0)

(val two   (lambda (f x) (f (f x)))
(val three (lambda (f x) (f (f (f x))))
(toInt (mult two three))

))))
```

Exercise 2: Lisp (from 2017's final)

Exercise 2.1: Scala implementation (5 points)

```scala
def derive(x: Symbol, expr: Any): Any = {
   expr match {
     case List('+, e1, e2) => List('+, derive(x, e1), derive(x, e2))
     case List('*, e1, e2) => List('+, List('*, derive(x, e1), e2), List('*, e1, derive(x, e2)))
     case _ => if (x == expr) 1 else 0
   }
 }
```

Exercise 2.2: Translation into Lisp (5 points)

```
(def (derive x expr)
  (if (isCons? expr)
```

```
    (if (= '+ (nth 0 expr))
      (list '+ (derive x (nth 1 expr)) (derive x (nth 2 expr)))
      (list '+ (list '* (derive x (nth 1 expr)) (nth 2 expr))
               (list '* (nth 1 expr) (derive x (nth 2 expr)))))
    (if (= x expr) 1 0))
  rest)
```

Exercise 3: Streams (from 2017's final)

```
def testStartsWith(input: Stream[Char], pattern: List[Char]): Option[Stream[Char]] = {
  pattern match {
    case Nil =>
      Some(input)
    case x :: xs =>
      input match {
        case y #:: ys if y == x =>
          testStartsWith(ys, xs)
        case _ =>
          None
      }
  }
}

def replaceAll(input: Stream[Char], pattern: List[Char],
    replacement: List): Stream[Char] = {
  testStartsWith(input, pattern) match {
    case None =>
      input match {
        case x #:: xs =>
          x #:: replaceAll(xs, pattern, replacement)
        case Stream.Empty =>
          Stream.Empty
      }

    case Some(rest) =>
      val newInput = replacement.toStream ++ rest
      replaceAll(newInput, pattern, replacement)
  }
}

def replaceAllMany(input: Stream[Char],
    patternsAndReplacements: List[(List[Char], List[Char])]): Stream[Char] = {
  val resultsOfTests = patternsAndReplacements.map {
    case (pat, repl) => testStartsWith(input, pat) -> repl
  }
  resultsOfTests.collectFirst {
    case (Some(rest), repl) =>
      val newInput = repl.toStream ++ rest
      replaceAllMany(newInput, patternsAndReplacements)
  }.getOrElse {
```

```
    input match {
      case x #:: xs =>
        x #:: replaceAll(xs, patternsAndReplacements)
      case Stream.Empty =>
        Stream.Empty
    }
  }
}
```

Exercise 4: The State Monad (from 2015's final)

This question is inspired from an example on the HaskellWiki. Check it out here:

https://wiki.haskell.org/State_Monad#Complete_and_Concrete_Example_1

Keeping score with mutable variables (1 point)

We first create a helper function, `scoreFunction`:

```
def scoreFunction(ga: GameAction, score: Int): Int = ga match {
  case EatMushroom    => score + 5
  case JumpOnTortoise => score + 10
  case SkidOnBanana   => score - 5
  case FallFromBridge => score - 10
}
```

We can then write the following imperative code:

```
def doAction2(ga: GameAction): String = {
  score = scoreFunction(ga, score)
  doAction(ga)
}
```

Desugaring For Comprehensions (2 points)

Doing the desugaring mechanically, we get:

```
acc flatMap { msg =>
  doAction2(elem) map { msg2 => msg ++ List(msg2) }
}
```

The monadic operations: unit (2 points)

```
def unit[A](a: A) = new StateM((s: Int) => (a, s))
```

The monadic operations: flatMap (3 points)

```
def flatMap[B](f: A => StateM[B]): StateM[B] = {
  val res = (s: Int) => {
    val (a, s2) = makeProgress(s)
    f(a).makeProgress(s2)
  }

  new StateM(res)
}
```

Actions that keep scores (2 points)

```
def doAction3(ga: GameAction): StateM[String] = for {
  score <- getState
  _ <- putState(scoreFunction(ga, score))
} yield doAction(ga)
```