Exercises marked * are more advanced. Please try all unstarred exercises before the tutorial meeting.

1. **Semantics of references**
   
   (a) Give explicit small-step rules for evaluating the sequential composition expression \( e_1; e_2 \). (Remember that it can also be viewed as syntactic sugar for \( \text{let } x = e_1 \text{ in } e_2 \) provided \( x \) is a fresh variable unused in either expression)
   
   (b) Evaluate the following expression to completion:
   
   \[
   \text{let } r = \text{ref}(\text{ref}(42)) \text{ in } !(!r))
   \]
   
   (c) Consider the following expression:
   
   \[
   \text{let } r = \text{ref}(\lambda x. x) \text{ in } r := (\lambda x. x + 1); (!r)(\text{true})
   \]
   
   Apply small-step evaluation to this expression until it reaches either a value or an error state.

2. **Interaction of references and evaluation order**

   Consider the following expression \( e \):
   
   \[
   \text{let } r = \text{ref}(42) \text{ in } (\lambda x. \text{print}(x); \text{print}(x)) \ (r := !r + 1; !r)
   \]
   
   where \( \text{print} \) is a side-effecting operation that fully evaluates its argument to a value and then prints it. For each of the following evaluation strategies, explain informally how \( e \) would be evaluated and what the printed output will be.

   (a) call-by-value
   
   (b) call-by-name
   
   (c) call-by-need / lazy evaluation

3. **Embedding LWhile in Scala**

   Recall the statements of LWhile:
   
   \[
   Stmt \ni s ::= \text{skip} | s_1; s_2 | x := e | \text{if } e \text{ then } s_1 \text{ else } s_2 | \text{while } e \text{ do } s
   \]
   
   In this exercise, we will show how to embed these statements into Scala, viewing LWhile’s variables as references using the Ref\[T\] type discussed in class:
class Ref[A](val x: A) {
  private var a = x
  def get = a
  def set(y: A) = { a = y }
}

Statements in LWhile will correspond to expressions of type Unit in Scala, and variables will correspond to instances of the Ref[T] type. Consider the following interface:

val skip : ()
def seq(s1: => Unit, s2: => Unit): Unit
def assign[T](x: Ref[T], e: => T): Unit
def ifthenelse(e: => Boolean, s1: => Unit, s2: => Unit): Unit
def whiledo(e: => Boolean, s: => Unit): Unit

Notice in particular that most arguments are passed by name (that is, their types are of the form => T).

(a) Implement the above operations.
(b) Why do the statements expressions in assign, ifthenelse, and whiledo need to be passed by name? What would happen if they were passed by value?
(c) (*) We have not considered how to adapt expressions. In LWhile, a mutable variable occurring in an expression is evaluated to its value. How should we adjust such expressions in LRef?

4. (*) Stream programming

Consider the following Stream type:

abstract class Stream[+A]
case object Empty extends Stream[Nothing]
case class SCons[+A](h: A, t: () => Stream[A]) extends Stream[A]

This defines a type of streams, which are similar to lists, but the evaluation of the tail of a stream is delayed.

Define Scala functions on streams as follows:

(a) const[A]: A => Stream[A] so that const(a) produces an infinite stream of a’s.
(b) take[A]: (Int,Stream[A]) => List[A] so that take(n,s) lists the first n elements from s.
(c) repeat[A]: (A => A) => A => Stream[A] such that

    repeat(a)(f) = Stream(a,f(a),f(f(a)),...)

For example, repeat(0) (incr) should produce the stream 0,1,2,3,..., if incr is the increment function.
(d) map[A]: Stream[A] => (A => B) => Stream[B] that applies the function f: A => B to each element of the stream s: Stream[A] yielding a stream of Bs.