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

1. Pairs, variants, and polymorphism in Scala

   Scala includes built-in pair types (T1, T2), with pairing written (e1, e2) and projection written e._1, e._2. Likewise, Scala’s library includes binary sums Either[T1,T2] with constructors Left(_). and Right(_). Pattern matching can be used to analyze Either[T1,T2]. Using these operations, write polymorphic Scala functions having the following types, polymorphic in A, B, C:

   (a)   (A, B) => (B, A)
   (b)   Either[A, B] => Either[B, A]
   (c)   ((A, B) => C) => (A => (B => C))
   (d)   (A => (B => C)) => ((A, B) => C)
   (e)   (Either[A, B] => C) => (A => C, B => C)
   (f)   (A => C, B => C) => (Either[A, B] => C)

2. Typing derivations

   Construct typing derivations for the following expressions, or argue why they are not well-formed:

   (a) \( \Lambda A. \lambda x : A. x + 1 \)
   (b) \( \lambda x : \text{int} \ + \text{bool}. \text{case} x \text{ of} \{ \text{left}(y) \Rightarrow y == 0 ; \text{right}(z) \Rightarrow z \} \)
   (c) \( \lambda x : \text{int} \times \text{int}. \text{if} \ x == \text{snd} \ x \text{ then } \text{left}(\text{fst} \ x) \text{ else } \text{right}(\text{snd} \ x) \)
   (d) \( \ast \ \Lambda A. \lambda x : A \times A. \text{if} \ x == \text{snd} \ x \text{ then } \text{fst} \ x \text{ else } \text{snd} \ x \)

3. Evaluation derivations

   Construct evaluation derivations for the following expressions, or explain why they do not evaluate:

   (a) \( (\Lambda A. \lambda x : A. x + 1)[\text{int}] 42 \)
   (b) \( (\Lambda A. \lambda x : A. x + 1)[\text{bool}] \text{ true} \)

4. Multiple argument functions

   So far, our function definitions take only one argument. Consider \( \text{LData} \) with named functions extended with multi-argument function definitions and applications:

   \[ e ::= \cdots \mid \text{let fun} \ f(x_1 : \tau_1, x_2 : \tau_2) = e_1 \text{ in } e_2 \mid f(e_1, e_2) \]
(a) Write appropriate typing rules for these constructs.
(b) Show that these constructs can be defined in L\text{Data}.
(c) What about functions of three or more arguments?

5. (⋆) Mutual recursion

In Lecture 5, we considered a simple form of recursion that just defines one recursive function with one argument. Part 4 of this tutorial showed how to accommodate multiple arguments. But what about mutual recursion?

A simple example is

\begin{verbatim}
let rec even(x:int) : bool = if x == 0 then true else odd(x - 1)
and odd(x:int) : bool = if x == 0 then false else even(x - 1)
in e
\end{verbatim}

Show that we can use pairing and rec to define these mutually recursive functions, by filling in the following template with an expression having type unit \(\rightarrow ((\text{int} \rightarrow \text{bool}) \times (\text{int} \rightarrow \text{bool}))\) with the desired behavior:

\begin{verbatim}
let p = \cdots in
let even = fst p() in
let odd = snd p() in
  e
\end{verbatim}