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 \((T_1, T_2)\), with pairing written \((e_1, e_2)\) and projection written \(e._1, e._2\). Likewise, Scala’s library includes binary sums \(\text{Either}[T_1, T_2]\) with constructors \(\text{Left}(\_\,)\) and \(\text{Right}(\_\,)\). Pattern matching can be used to analyze \(\text{Either}[T_1, T_2]\). Using these operations, write Scala functions having the following types, polymorphic in \(A, B, C\):

   - (a) \((A, B) \Rightarrow (B, A)\)
   - (b) \(\text{Either}[A, B] \Rightarrow \text{Either}[B, A]\)
   - (c) \(((A, B) \Rightarrow C) \Rightarrow (A \Rightarrow (B \Rightarrow C))\)
   - (d) \((A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A, B) \Rightarrow C)\)
   - (e) \((\text{Either}[A, B] \Rightarrow C) \Rightarrow (A \Rightarrow C, B \Rightarrow C)\)
   - (f) \((A \Rightarrow C, B \Rightarrow C) \Rightarrow (\text{Either}[A, B] \Rightarrow C)\)

2. **Typing derivations**

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

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

3. **Lists**

   We could add built-in lists to \(L_{\text{Data}}\) as follows:

   \[
   \begin{align*}
   e &::= \cdots | \text{nil} | e_1 :: e_2 | \text{case list } e \text{ of } \{ \text{nil} \Rightarrow e_1 ; x :: y \Rightarrow e_2 \} \\
   v &::= \cdots | \text{nil} | v_1 :: v_2 \\
   \tau &::= \cdots | \text{list}[\tau]
   \end{align*}
   \]

   Define \(L_{\text{List}}\) to be \(L_{\text{Data}}\) extended with the above constructs.

   The typing rule for \(\text{case list}\) is:

   \[
   \Gamma \vdash e : \text{list}[\tau] \quad \Gamma \vdash e_1 : \tau' \quad \Gamma, x : \tau, y : \text{list}[\tau] \vdash e_2 : \tau' \\
   \Gamma \vdash \text{case list } e \text{ of } \{ \text{nil} \Rightarrow e_1 ; x :: y \Rightarrow e_2 \} : \tau'
   \]

   The basic idea here is: Given a list \(e\), a \(\text{case list}\) expression does a case analysis. If \(e\) evaluates to \(\text{nil}\), then we evaluate \(e_1\). Otherwise, \(e\) must evaluate to a non-empty list of the form \(v :: v'\), and we bind \(x\) to the head element \(v\) and \(y\) to the tail \(v'\), and evaluate \(e_2\).
(a) Write appropriate typing rules for \texttt{nil} and ::.
(b) \(\star\) Write appropriate evaluation rules for the above constructs.

4. \(\star\) Multiple argument functions and mutual recursion

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

\[
e ::= \cdots | \text{let fun } f(x_1 : \tau_1, x_2 : \tau_2) = e_1 \text{ in } e_2 | f(e_1, e_2)
\]

i. Write appropriate typing rules for these constructs.
ii. Show that these constructs can be defined in \(L_{\text{Data}}\).
iii. What about functions of three or more arguments?

(b) 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

\[
\begin{align*}
\text{let rec even}(x : \text{int}) : \text{bool} &= \text{if } x == 0 \text{ then true else odd}(x - 1) \\
\text{and odd}(x : \text{int}) : \text{bool} &= \text{if } x == 0 \text{ then false else even}(x - 1) \\
\text{in } e
\end{align*}
\]

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

\[
\begin{align*}
\text{let } p &= \cdots \text{ in} \\
\text{let even } &= \text{fst } p() \text{ in} \\
\text{let odd } &= \text{snd } p() \text{ in} \\
\text{e}
\end{align*}
\]