Topic 1. Scoping vs. value binding. When thinking about variables, don’t confuse scoping and value binding. Scoping is a static concept, and refers to the region of code where a particular variable can be referred to as you’re writing the code. Value binding is a dynamic concept, and refers to the value associated with a variable at run time. So for example, the scope of \texttt{var} in \texttt{let var = 2 * 3 in var + 20} is \texttt{var + 20}, while the value binding of \texttt{var} during execution of the program is 6.

Topic 2. Values evaluation. Every expression has an evaluation rule; values always evaluate to themselves. Since we view functions as values, this means functions evaluate to themselves.

Topic 3. Function application. Function use and application are the same thing. The semantics of function application is simple but powerful; evaluate \( (\text{fun } x \to e)\ v \) by binding \( x \) (the formal parameter) to the value \( v \) (the argument), and then evaluate \( v \). So for example \( (\text{fun } x \to 2 \times x)10 \Downarrow 20 \), since we evaluate the application by evaluating \( 2 \times x \) with \( x \) bound to 10, which is equivalent to evaluating \( 2 \times 10 \).

Topic 4. Call-by-name vs. call-by-value. When evaluating an application \( (\text{fun } x \to e)\ e' \) where \( e' \) is not a value, we can either specify to bind \( x \) to \( e' \) for the evaluation of \( e \), or we can specify to bind \( x \) to \( v \) for the evaluation of \( e \), where \( e' \Downarrow v \). That is, we can either just pass in the argument as is, or evaluate it before passing it in. The former strategy is called \textit{call-by-name}, and the latter is called \textit{call-by-value}. OCaml, like most modern languages, is call-by-value.

Topic 5. Recursive declarations. OCaml provides recursive type declarations of the form \texttt{let rec x = e1 in e2}, where the scope of \( x \) is \( e1 \) and \( e2 \)– that is, \( x \) may be self-referential. The main purpose of this declaration form is to define recursive functions. Example:

\[
\begin{align*}
\text{let rec expt} & = (\text{fun } x \to \text{if } x = 0 \text{ then } 1 \text{ else } 2 \times \text{expt}(x - 1)) \\
\text{let rec fact} & = (\text{fun } x \to \text{if } x = 0 \text{ then } 1 \text{ else } x \times \text{fact}(x - 1))
\end{align*}
\]

Note that in each of these function definitions, there are cases in which a recursive call is made; for both \texttt{fact}, \texttt{expt}, this is in case \( x > 0 \). There is also a case in which the function returns immediately; for both \texttt{fact}, \texttt{expt}, this is in case \( x = 0 \). We call the latter the \textit{basis of recursion}.

Recursive case: A case in which the argument to a recursively defined function causes a recursive call to be made.

Base case: A case in which the argument to a recursively defined function causes the return of a non-recursive computation.

Note that without a basis, a recursive function will \textit{diverge}:

\[
\begin{align*}
\text{let rec diverge} & = (\text{fun } x \to x + \text{expt}(x + 1)) \text{ in diverge 0}
\end{align*}
\]