Topic 1. Tuple types. Composite types are datatypes that are built up from other datatypes. Perhaps the simplest example of composite types in OCaml are tuples; OCaml tuples are constructed with a comma (,), and can be any length, and heterogeneous in component type. Tuple types forms are built up from the type of components, and the * type constructor:

\[(1, 2) : \text{int} * \text{int}\]
\[(1 + 2, \text{"hello"}, (\text{fun } x \rightarrow x + 1)) : \text{int} * \text{string} * (\text{int} \rightarrow \text{int})\]

Topic 2. Composite type construction and destruction. Any composite datatype has a means to construct expressions in the datatype, and a means to destruct expressions— that is, to build them up and take them apart. Otherwise, they are not useful. Pairs (2-tuples) are constructed with the comma, and destructed with the fst and snd operator.

Topic 3. Pattern matching. Pattern matching also serves as a means to destruct datatypes, by providing a means to bind individual variables to components of composite types. Patterns can also include constants, that can help define cases in pattern matching clauses.

\[
\begin{align*}
\text{let } (x, y) &= (1, 2) \text{ in } x + y &\Rightarrow 3 \\
\text{match } (1, 2, 3) \text{ with } (0, x, y) &\rightarrow x * y \\
&| (1, x, y) &\rightarrow x + y \\
&| (1, x, y) &\rightarrow x * y &\Rightarrow 5 \\
\end{align*}
\]

\[
\text{let rec } \text{fact } n = \text{match } n \text{ with } \\
0 &\rightarrow 1 \\
| n &\rightarrow n * \text{fact}(n-1) \\
\]

Topic 4. Functions of more than one argument. The view of tuples as values, together with pattern matching, allows definition of functions of “more than one argument” with the existing machinery for function definitions, which only allow one argument (which may be a tuple):

\[
\begin{align*}
\text{let } \text{add } (x, y) &= x+y \text{ in } \text{add}(3, 4) &\Rightarrow 7 \\
\end{align*}
\]