Topic 1. **Match redundancy and exhaustiveness.** Pattern matching clauses are evaluated by matching the valuation of the given expression against the given patterns, in a top-to-bottom order. Since more than one pattern can match any given value, care must be taken to avoid redundancy in match clauses; for example, the following incorrectly defined factorial function diverges on every argument:

```ocaml
let rec fact n =  
  match n with  
  _ -> n * fact(n-1)  
| 0 -> 1
```

Another issue is exhaustiveness; some patterns will not match every form of a given expression’s domain type. However, this may be acceptable if you don’t expect to be given values other than the specified form—just be sure to define this form in the comment.

Topic 2. **Patterns, names, and scoping.** Patterns allow introduction of several variables at binding points, with the usual scoping rules in effect. For example, the scope of x and y in:

```ocaml
let (x,y) = (1, (2+3)) in (x * y) + 7
```

is the expression \((x * y) + 7\). Match expressions are a new way of declaring variables, hence they have their own scoping rules. For any expression of the form:

```ocaml
match e with  
  p1 -> e1  
| ...  
| pn -> e2
```

the scope of the variables declared in each pattern \(p1\) through \(pn\) are the expressions \(e1\) through \(en\), respectively. Hence, the variable \(y\) is used out-of-scope in the following function declaration, that will be rejected by the compiler:

```ocaml
let f x =  
  match x with  
  (0,y) -> y  
| (1,_) -> y * 2
```

Topic 3. **Type polymorphism.** When a function manipulates compound types, its actions may not depend on the specific types of the components, only the “skeleton” of the compound type. This allows the function to be defined abstractly with respect to the component types, which is reflected in a polymorphic type of the function. A polymorphic type can be used at several different types. For example, consider:

```ocaml
let thrd (_,_,x) = x
```

The function \(\text{thrd}\) clearly takes 3-tuples \((e1, e2, e3)\) as arguments, and returns the third element \(e3\). But observe that \(\text{thrd}\) only requires its argument to be a 3-tuple; it doesn’t care about the particular type of \(e1, e2,\) or \(e3\). This is reflected in its type:
\[
\text{thrd} : (\text{`a} * \text{`b} * \text{`c}) \to \text{`c}
\]

Here, `a, `b, `c are type variables, and range over any type. Intuitively, thrd can “assume” any type obtainable by consistent substitution for these variables, for example:

\[
\begin{align*}
\text{thrd} : (\text{int} * \text{int} * \text{int}) & \to \text{int} \\
\text{thrd} : ((\text{int} * \text{float}) * \text{string} * \text{bool}) & \to \text{bool} \\
\text{thrd} : (\text{int} * \text{int} * (\text{`a} \to \text{`a})) & \to (\text{`a} \to \text{`a})
\end{align*}
\]

which typings respectively allow typing of the following expressions:

\[
\begin{align*}
\text{thrd} (1,2,3) & \quad (* \text{evaluates to } 3 *) \\
\text{thrd} ((1,2.0),''hi'', \text{false}) & \quad (* \text{evaluates to } \text{false } *) \\
\text{thrd} (1,2,(\text{fun } x \to x)) & \quad (* \text{evaluates to } (\text{fun } x \to x) *)
\end{align*}
\]

Hence, polymorphism allows functions to be defined in a general manner, which is why polymorphic types are sometimes called generics (e.g. in Java). C++ templates wish they were polymorphic types.