Topic 1. Terminology. Computer programs exhibit both dynamic and static properties:

- **Dynamic**: run-time properties
- **Static**: compile-time/syntactic properties

For example, non-termination is a dynamic property, while well-typedness of e.g. programs is a static property. Static properties are determined directly from program syntax, before any evaluation of the program takes place.

Topic 2. OCaml. OCaml programs are defined in terms of expressions $e$. Every expression $e$:

- Has a type $e : \tau$, otherwise rejected (may be unsafe!). Types are reconstructed via a static analysis.
- May evaluate to a value $e \Downarrow v$, unless it does not terminate, in which case we write $e \Uparrow$.
  - (NB: all values $v$ are expressions $e$)
- May have a side effect (e.g. I/O, mutation of state)

For the first few lectures, we will focus on effects-free fragment of OCaml; in general, we will not be concerned with I/O, focusing instead on the definition and manipulation of data types. This means that we will be considering functional programming, versus procedural programming where manipulation of state is the focus of programming technique.

Topic 3. Types. A type is:

- A set of values
- A set of operations on values of that type

For example, 1 is a value of type \texttt{int}, whereas + is an operation on \texttt{ints}. Compound expressions are assigned the type of value to which the expression will evaluate; for example, $(1 + 3) : \texttt{float}$. Although $1 + 3$ is not a value, it will evaluate to a value 4. Keep in mind the following principle:

Types statically predict the class of values to which a given expression will evaluate

Note that this does not say which value, just the class of values– for example, the type of $1 + 3$ is not \texttt{int}! There’s only so much you can predict without running the code...

Also, keep in mind that typing of an expression is based on typing of its subexpressions. For example, $e_1 + e_2 : \texttt{float}$ iff $e_1 : \texttt{float}$ and $e_2 : \texttt{float}$.

Topic 4. Evaluation. Evaluation is a formalization of computation. With well-designed programming languages, we can talk about the evaluation behavior of programs without worrying about the particular machine that code will run on. For example, any expression $e_1 + e_2 \Downarrow n$ iff $e_1 \Downarrow n_1$ and $e_2 \Downarrow n_2$ and $n = “n_1 + n_2”$, where quotation marks mean that I’m quoting the “true mathematical universe”. In other words, evaluation of an expression is based on evaluation of its subexpressions– for simple base types, anyway.

Think of evaluation as a well-specified algebra. For example, from algebra you know that if $a = (1 + 2) \times (3 + 2)$, then $a = 15$; the evaluation relation makes explicit in what order (how) the various parts of the equation should be evaluated; first 1+2, then 3+2, then 3 * 5.
Topic 5. Errors. Type errors indicate problems with the way you’re trying to combine expressions, and occur statically:

“hello” * true ill-typed

Not all semantic, or programmer, errors can be predicted by the type system. However, well-typedness ensures that programs won’t encounter the sorts of errors where the program crashes, where evaluation is ill-defined. For example, you will never encounter a segmentation fault when programming in Ocaml.