**Topic 1. Lists.** Lists are ordered sequences of heterogeneous data. The type of lists always reflects the type of its elements:

\[
\begin{align*}
[2,46] & : \text{int list} \\
[(3,4.0);(2,6.1);(5,8.9)] & : \text{(int * float) list} \\
[[3;5;3];[2;1];[]] & : \text{int list list}
\end{align*}
\]

The \textit{cons} notation :: serves double-duty, to construct lists:

\[
2::3::5::[] \Downarrow [2;3;5]
\]

and also to destruct lists, by use in a pattern:

\[
\text{let } (x::xs) = [2;3;5] \text{ in } x \Downarrow 2
\]

Note that cons always takes a single list element on the left, and a list on the right; naturally, the type of the element must agree with the type of the list, for example \(2::['a']\) is not well-typed. The \textit{empty list} is denoted \([]\). A list is recursively defined datastructure; any list \(l\) is either empty \([]\) (the base case), or a cons \(v::l'\) of a single element \(t\) and another list \(l'\) (the recursive case; note that lists \(l\) are defined in terms of smaller lists \(l'\) in this case).

**Topic 2. Higher order functions.** Higher order functions are functions that take other functions as arguments, and return functions as results. For example, functional composition can be defined in OCaml as follows:

\[
\text{let compose } (f,g) = (\text{fun } x -> f(g(x)))
\]

which could be used as follows:

\[
\text{let add1 } x = x + 1 \\
\text{let add2 } = \text{compose add1 add1} \\
\text{add1 } 2 \Downarrow 3 \\
\text{add2 } 2 \Downarrow 4
\]

Observe that the higher order nature of compose is reflected in its type, which specifies that it takes a pair of function types as arguments:

\[
\text{compose : } (('a \rightarrow 'b) \times ('c \rightarrow 'a)) \rightarrow 'c \rightarrow 'b
\]

**Topic 3. Curried vs. uncurried style.** Higher order functions give us another way of defining functions of more than one argument. For example, we could define a plus function, that adds two integer arguments, in a couple of ways:

\[
\text{let plus } (x,y) = x + y \ (* \text{uncurried style} *) \\
\text{let plus' } = (\text{fun } x -> \text{fun } y -> x + y) \ (* \text{curried style} *) \\
\text{let plus' } x y = x + y \ (* \text{curried style with syntactic sugaring} *)
\]
The curried style is useful in case you want to apply the function to just one of its arguments (a partial application):

```latex
let incr = plus’ 1
incr 2 \rightarrow 3
incr 3 \rightarrow 4
```

“Currying” is named after the great Computer Scientist and Programming Language theorist Haskell Curry, who studied the phenomena in the mid-20th century.

**Topic 4. Abstracting patterns of control.** Higher order programming techniques allow us to abstract patterns of control over datastructures, eliminating code duplication and capturing important programming patterns in a precise manner. For example, suppose we defined separate functions for doubling all elements of an integer list, and for turning all elements of an integer list into floats:

```ocaml
let rec double_all l = match l with
  | [] -> []
  | x::xs -> (2 * x) :: (double_all xs)

double_all : int list -> int list
```

```ocaml
let rec float_all l = match l with
  | [] -> []
  | x::xs -> (float x) :: (float_all xs)

float_all : int list -> float list
```

Observe how similar these functions are; at a higher level of abstraction, each is applying some transformation to every element of a given list. This pattern can be captured with the function `map`:

```ocaml
let rec map f l = match l with
  | [] -> []
  | (x::xs) -> (f x) :: (map f xs)

map : ('a -> 'b) -> 'a list -> 'b list
```

mapping a function across a list transforms each element of the list accordingly:

```ocaml
map (fun x -> 2 * x) [1;2;3;4] \rightarrow [2;4;6;8]
map (fun x -> float x) [1;2;3;4] \rightarrow [1.0;2.0;3.0;4.0]
```

In conjunction with partial application:

```ocaml
let double_all = map (fun x -> 2 * x)
let float_all = map float
```

We can use pattern matching to really get some elegant, powerful stuff going on. For example, imagine that we represent a graph as a list of pairs of \((x, y)\) coordinates:

```ocaml
let graph = [(1.0, 3.5);(2.2,4.6);(4.8,9.2)]
```

We can use `map` in conjunction with partial application and pattern matching that extracts the \(y\) coordinates from any graph:

```ocaml
let xcoords = map (fun (x,y) -> y)
xcoords graph \rightarrow [1.0;2.2;4.8]
```