Topic 1. Variable declarations. Declarations introduce names for values. A binding environment associates names with values. In an expression `let x = e1 in e2`, the name `x` is introduced as a name for the valuation of `e1` in the evaluation of `e2`. Note very carefully that `x` is not a mutable variable, it is just a name for a value; keep in mind the saying: *variables don’t vary.*

Topic 2. Scope. The scope of `x` in a declaration `let x = e1 in e2` is `e2`; this is the only region of code where the naming of `e2` as `x` can be referred to. Outside of the expression, `x` is not in scope. So for example, the expression `let num = 2 in num + num` is sensible, but the expression `(let num = 2 in num + num) * num` is not, since the rightmost occurrence of `num` is not in the scope of its declaration. Names can be rebound as often as you like; if a rebinding of the same name occurs within its own scope, e.g. `let x = 1 in let x = false in not x`, the “most recent” binding always shadows the previous binding. Example:

```
let x = 2 in ((let x = x + x in 2 * x) + x) ⇓ 10
```

Any expression that introduces variable names is subject to scoping rules, such as functions, discussed below.

Topic 3. Top-level declarations. The OCaml compiler allows declarations of the form `let x = e`, but only at the top level for introduction of global variables. Expression `let x = e1 in e2` can be used to declare variables anywhere expressions are allowed.

Topic 4. Functions. The essence of functional programming is the treatment of functions as values. Any function `fun x -> e` is a value. Functions have types of the form `τ₁ → τ₂`, where `τ₁` is the type of the domain, `τ₂` is the type of the range. The scope of `x` in `fun x -> e` is `e`. To type check functions `(fun x -> e)

1. extend environment with `x : τ`, where `τ` is the domain type
2. type check `e2`, yielding `e2 : τ'`
3. remove `x : τ` from environment, yield `(fun x -> e) : τ → τ'`

Functions are used by applying them, in expressions of the form `e₁ e₂`, where `e₁` evaluates to a function.

- given an application `e₁ e₂`, we have `e₁ e₂ : τ` iff `e₁ : τ' → τ` and `e₂ : τ'`

Examples:

```
let x = 3 in
(fun y -> x + y) 2 ⇓ 5

let x = 3 in
(let x = 2 in (fun y -> x + y)) 2 ⇓ 4
```