

Introduction to Functional Programming in *OCaml*

Roberto Di Cosmo, Yann Régis-Gianas, Ralf Treinen

Week 6 - Sequence 0: Structuring software with modules



Programming in the large

- ▶ So far, we have seen how to define **oplevel** types, functions and basic values.
- ▶ We have **programmed in the small**, defining data structures and algorithms.
- ▶ *OCaml core language* is great to write them in a safe, sound and efficient way.
- ▶ In a large project, one difficulty is to manage the high number of definitions.
- ▶ To not be lost into the implementation details, **abstraction** is the key.
- ▶ An **abstraction** is a concept that can be understood intrinsically, without a precise knowledge of its implementation.
- ▶ An abstraction can be built on top of other abstractions.
- ▶ Good architectures are made of layers of abstractions.

The rules of the game are different

- ▶ The program must be **divided into components**.
- ▶ Identifiers must be organised to avoid **naming conflicts**.
- ▶ The **layers of abstraction** must be **enforced**.
- ▶ **Glueing components** together should be feasible **after their development**.

The module language of *OCaml*
fulfills all these requirements!

Overview of Week 6

1. Structuring software with modules
2. Information hiding
3. Case study: An abstract type for dictionaries
4. Functors
5. Modules as compilation units

Module as a namespace

- ▶ We have seen that the dot-notation can be used to access a module component.
- ▶ `List.length` refers to the `length` function of the module `List`.
- ▶ If you want to avoid writing “`List.`”, it is possible to **open the namespace** of the module `List` by writing “**open** `List`”
- ▶ After that command, `length` implicitly refers to `List.length`.
- ▶ If two modules contain two identical identifiers, the definition from the last opened module is used.

Implementing a module

- ▶ To define a module:

```
module SomeModuleIdentifier =  
  struct  
    (* a sequence of definitions *)  
  end
```

- ▶ The identifier of a module must start with an uppercase letter.
- ▶ A module contains value, type and exception definitions.
- ▶ A module can be aliased:

```
module SomeModuleIdentifier =  
  SomeOtherModuleIdentifier
```

A module providing a stack data structure I

```
module Stack = struct
  type 'a t = 'a list
  let empty = []
  let push x s = x :: s
  let pop = function
    | [] -> None
    | x :: xs -> Some (x, xs)
end;;

# module Stack :
sig
  type 'a t = 'a list
  val empty : 'a list
  val push : 'a -> 'a list -> 'a list
  val pop : 'a list -> ('a * 'a list) option
end
```

A module providing a stack data structure II

```
let s = Stack.empty;;  
# val s : 'a list = []  
let s = Stack.push 1 s;;  
# val s : int list = [1]  
let x, s =  
  match Stack.pop s with  
  | None -> assert false  
  | Some (x, s) -> (x, s);;  
# val x : int = 1  
val s : int list = []  
let r = Stack.pop s;;  
# val r : (int * int list) option = None
```


Module signatures

- ▶ The type of a module is called a **signature** or an **interface**.
- ▶ As we have seen on the previous example, ***OCaml infers signatures***.
- ▶ The programmer can force a module to have a specific signature.
- ▶ Publishing **well-designed** signatures is a very important communication aspect in a large project, this is the topic of the next sequence.
- ▶ A signature has the following shape:

```
sig
  (* A sequence of declarations of the form: *)
  val some_identifier : some_type
  type some_type_identifier = some_type_definition
  exception SomeException of some_type
end
```

Hierarchical structures of modules

- ▶ A module can also contain module definitions.
- ▶ A signature can also contain module signatures.
- ▶ If the module B is defined inside module A, “A.B.” is the path to its namespace.
- ▶ It is forbidden to define two submodules with the same name in a module.

A submodule for trees in a module for forests I

```
module Forest = struct
  type 'a forest = 'a list
  module Tree = struct
    type 'a tree = Leaf of 'a | Node of 'a tree forest
  end
end;;

# module Forest :
sig
  type 'a forest = 'a list
  module Tree :
    sig
      type 'a tree = Leaf of 'a | Node of 'a tree forest
    end
  end
end
```

A submodule for trees in a module for forests II

```
open Forest.Tree  
let t = Leaf 42;;  
# val t : int Forest.Tree.tree = Leaf 42
```