

Programmation Fonctionnelle (PF)

INFO4

Cours 9 et 10 : modules, compilation et foncteurs

Jean-François Monin, Benjamin Wack



2019 - 2020

Plan

Modules

- Motivations

- En pratique

Compilation

- Principes de base

- Compilation séparée

- Makefile

Foncteurs

Plan

Modules

Motivations

En pratique

Compilation

Principes de base

Compilation séparée

Makefile

Foncteurs

Objectifs de la modularité

- ▶ Faciliter la réutilisation de code

Objectifs de la modularité

- ▶ Faciliter la réutilisation de code
- ▶ Organiser le développement : **indépendance** entre fournisseur et utilisateur
 - ▶ rôle pivot de l'*interface* = joue le rôle de contrat
 - ▶ l'utilisateur n'a pas besoin de connaître les détails de la réalisation
 - ▶ le fournisseur se concentre sur les fonctions promises

Objectifs de la modularité

- ▶ Faciliter la réutilisation de code
- ▶ Organiser le développement : **indépendance** entre fournisseur et utilisateur
 - ▶ rôle pivot de l'*interface* = joue le rôle de contrat
 - ▶ l'utilisateur n'a pas besoin de connaître les détails de la réalisation
 - ▶ le fournisseur se concentre sur les fonctions promises
- ▶ Notion de *type abstrait*
 - ▶ l'utilisateur ne peut pas exploiter les détails de la réalisation
 - ▶ garantie des *invariants* : il suffit que chaque fonction de l'interface respecte ces invariants
 - ▶ possibilité de changer de représentation interne sans remettre en cause le code de l'utilisateur

Les modules OCaml

Réalisation

- ▶ interface = signature S qui liste les types et fonctions fournis
- ▶ implémentation regroupée dans un module $M : S$
- ▶ notation pointée $M.f$

Les modules OCaml

Réalisation

- ▶ interface = signature S qui liste les types et fonctions fournis
- ▶ implémentation regroupée dans un module $M : S$
- ▶ notation pointée $M.f$

Bénéfices immédiats

- ▶ Compatible avec la discipline de typage fort habituelle
- ▶ Espace de noms organisé : `Liste.app` et `Arbre.app` au lieu de `app_liste` et `app_arbre`
- ▶ Prépare l'orienté objet
- ▶ Possibilité de définir des sous-modules ($M.N.f$)

Remarque : il existe des modules similaires en Coq.

Plan

Modules

Motivations

En pratique

Compilation

Principes de base

Compilation séparée

Makefile

Foncteurs

Déclaration

```
module Nomdumodule =  
  struct  
    type t = A | B of int  
    type u = char  
    ...  
    let f x = if x=A then (B 0) else A  
    let g x = x  
    ...  
end
```

Utilisation

Notation pointée

```
f (B 10);;
```

```
Error: Unbound value f
```

Utilisation

Notation pointée

```
f (B 10);;
```

```
Error: Unbound value f
```

```
Nomdumodule.f (Nomdumodule.B 10);;
```

```
- : Nomdumodule.t = Nomdumodule.A
```

Peut devenir lourd si le nom du module est long...

Utilisation

Notation pointée

```
f (B 10);;
```

```
Error: Unbound value f
```

```
Nomdumodule.f (Nomdumodule.B 10);;
```

```
- : Nomdumodule.t = Nomdumodule.A
```

Peut devenir lourd si le nom du module est long...

Instruction open

```
open Nomdumodule
```

```
f (B 10);;
```

```
- : Nomdumodule.t = A
```

Mais **aplatit l'espace de noms** : risque d'écraser une valeur

Signature

Inférée lors de la définition du module

```
module Nomdumodule :  
  sig  
    type t = A | B of int  
    type u = char  
    ...  
    val f : t -> t  
    val g : 'a -> 'a  
    ...  
  end
```

Signature *

Ou spécifiée pour définir l'interface visible

```
module type NOMDELASIG =  
  sig  
    type t  
    type u = char  
    ...  
    val g : t -> t  
    ...  
end
```

Signature *

Ou spécifiée pour définir l'interface visible

```
module type NOMDELASIG =  
  sig  
    type t  
    type u = char  
    ...  
    val g : t -> t  
    ...  
  end  
  
module Nomdumodule : NOMDELASIG = struct ... end
```


Signature *

Ou spécifiée pour définir l'interface visible

```
module type NOMDELASIG =  
  sig  
    type t  
    type u = char  
    ...  
    val g : t -> t  
    ...  
  end  
  
module Nomdumodule : NOMDELASIG = struct ... end
```

La fonction `f` est **inaccessible** depuis l'extérieur si elle n'apparaît pas dans la signature.

Signature *

Ou spécifiée pour définir l'interface visible

```
module type NOMDELASIG =  
  sig  
    type t (* abstrait *)  
    type u = char (* concret *)  
    ...  
    val g : t -> t  
    ...  
  end  
  
module Nomdumodule : NOMDELASIG = struct ... end
```

La fonction `f` est **inaccessible** depuis l'extérieur si elle n'apparaît pas dans la signature.

Signature *

Ou spécifiée pour définir l'interface visible

```
module type NOMDELASIG =  
  sig  
    type t (* abstrait *)  
    type u = char (* concret *)  
    ...  
    val g : t -> t (* polymorphisme bridé *)  
    ...  
  end  
  
module Nomdumodule : NOMDELASIG = struct ... end
```

La fonction `f` est **inaccessible** depuis l'extérieur si elle n'apparaît pas dans la signature.

Exemple : à vous de jouer

Écrire un module pour des multi-ensembles : que faut-il définir ?

Exemple : à vous de jouer

Écrire un module pour des multi-ensembles : que faut-il définir ?

- ▶ Interface

Exemple : à vous de jouer

Écrire un module pour des multi-ensembles : que faut-il définir ?

- ▶ Interface

Par exemple : ensemble vide, insérer, compter, retirer

Exemple : à vous de jouer

Écrire un module pour des multi-ensembles : que faut-il définir ?

- ▶ Interface

Par exemple : ensemble vide, insérer, compter, retirer

- ▶ Réalisation

Exemple : à vous de jouer

Écrire un module pour des multi-ensembles : que faut-il définir ?

- ▶ Interface

Par exemple : ensemble vide, insérer, compter, retirer

- ▶ Réalisation

représentation interne + fonctions

Type abstrait

- ▶ Quelle partie du module `MultiEns` devrait-on masquer ?

Type abstrait

- ▶ Quelle partie du module `MultiEns` devrait-on masquer ?

La représentation interne

Type abstrait

- ▶ Quelle partie du module `MultiEns` devrait-on masquer ?

La représentation interne

- ▶ Comment ?

Type abstrait

- ▶ Quelle partie du module MultiEns devrait-on masquer?

La représentation interne

- ▶ Comment ?

```
module type MULTIENS = sig
  type 'a t
  val vide : unit -> 'a t
  val inserer : 'a -> 'a t -> 'a t

  val compter : 'a -> 'a t -> int
  val retirer : 'a -> 'a t -> 'a t
  exception Element_absent
end
```

Puis on implémente

```
module MultiEns : MULTIENS = struct
  type 'a t = 'a list
  let vide = ...
  let inserer = fun x e → ...

  let rec compter = fun x e → ...

  exception Element_absent
  let rec retirer = fun x e → ...
end
```

Et enfin on utilise

```
module M = MultiEns
```

```
# let present = fun x e → ...
```

```
val present : 'a -> 'a M.t -> bool = <fun>
```

```
# let rec eliminer = fun x e → ...
```

```
val eliminer : 'a -> 'a M.t -> 'a M.t = <fun>
```

Plus tard on peut changer d'avis...

```
module MultiEns : MULTIENS = struct
  type 'a t = ('a * int) list

  let vide = fun () → ...

  let inserer = fun x e → ...

  ...
end
```

... sans changer le programme précédent !

Plan

Modules

Motivations

En pratique

Compilation

Principes de base

Compilation séparée

Makefile

Foncteurs

Un fichier source OCaml, trois possibilités

Interpréteur interactif

- ▶ adapté pour le développement, pas pour la production
- ▶ complique l'utilisation de bibliothèques

Un fichier source OCaml, trois possibilités

Interpréteur interactif

- ▶ adapté pour le développement, pas pour la production
- ▶ complique l'utilisation de librairies

`ocamlc`

produit du bytecode :

- ▶ indépendant de la machine (*portable*)
- ▶ interprété par une machine virtuelle
- ▶ susceptible d'être utilisé avec `ocamldebug`

Un fichier source OCaml, trois possibilités

Interpréteur interactif

- ▶ adapté pour le développement, pas pour la production
- ▶ complique l'utilisation de librairies

`ocamlc`

produit du bytecode :

- ▶ indépendant de la machine (*portable*)
- ▶ interprété par une machine virtuelle
- ▶ susceptible d'être utilisé avec `ocamldebug`

`ocamlopt`

produit du code natif (dépendant de la machine) donc *efficace*.

Hello world*

Les doubles point-virgules sont facultatifs (voire déconseillés) dans un code source.

Un exemple_compil.ml à compiler

```
let x=42
let main =
  print_string "Hello World! \n";
  print_int(x);
  print_newline()
```

```
$ ocamlc exemple_compil.ml -o mon_exec
```

```
$ ./mon_exec
```

```
Hello World!
```

```
42
```

Compilation vers du bytecode

Le code source ne doit pas contenir de directives du toplevel (`#...`).

`ocamlc file.ml` produit :

- ▶ `file.cmi`
- ▶ `file.cmo` (fichier *objet*)
(peut être chargé dans un toplevel OCaml)
- ▶ et un “exécutable” (précisé par `-o ...` sinon `a.out`)
(nécessite `ocamlrun` pour s'exécuter)

Compilation vers du bytecode

Le code source ne doit pas contenir de directives du toplevel (`#...`).

`ocamlc file.ml` produit :

- ▶ `file.cmi`
- ▶ `file.cmo` (fichier *objet*)
(peut être chargé dans un toplevel OCaml)
- ▶ et un “exécutable” (précisé par `-o ...` sinon `a.out`)
(nécessite `ocamlrun` pour s'exécuter)

Options possibles :

- ▶ `-I ...` donne le chemin des fichiers à inclure
- ▶ `-g` ajoute des infos de debug pour `ocamldebug`
- ▶ ...

Compilation vers du code natif

ocamlopt file.ml produit :

- ▶ file.cmi
- ▶ file.cmx (édition de liens)
- ▶ file.o (fichier *objet*)
- ▶ et un exécutable (-o ... sinon a.out)
(standalone)

Plan

Modules

Motivations

En pratique

Compilation

Principes de base

Compilation séparée

Makefile

Foncteurs

Modules et fichiers

- ▶ cohérence module-fichier :
 - ▶ `fichier.ml` correspond au module `Fichier`
 - ▶ `fichier.mli` correspond à la signature `Fichier`
- ▶ possibilité de déclarer des (sous-)modules et des (sous-)signatures dans un fichier mais on s'y perd vite !

Attention aux **majuscules**

Les noms de modules et de signatures commencent par des majuscules (même si le nom du fichier est en minuscules!).

Exemple avec 2 fichiers

Fichier signe.ml

```
type t = Neg | Zer | Pos
let f x = if x < 0 then Neg else if x = 0 then Zer else Pos
```

Fichier arith.ml

```
let reduit x = match Signe.f x with
  | Signe.Neg -> -1 | Signe.Zer -> 0 | Signe.Pos -> 1
```

Exemple avec 2 fichiers

Fichier signe.ml

```
type t = Neg | Zer | Pos
let f x = if x < 0 then Neg else if x = 0 then Zer else Pos
```

Fichier arith.ml

```
let reduit x = match Signe.f x with
  | Signe.Neg -> -1 | Signe.Zer -> 0 | Signe.Pos -> 1
```

Version « équivalente » en un seul fichier

```
module Signe = struct
  type t = Neg | Zer | Pos
  let f x = if x < 0 then Neg else if x = 0 then Zer else Pos
end
let reduit x = match Signe.f x with
  | Signe.Neg -> -1 | Signe.Zer -> 0 | Signe.Pos -> 1
```

Avec fichier d'interface .mli

Fichier signe.mli

```
type t = Neg | Zer | Pos  
val f : int -> t
```

Avec fichier d'interface .mli

Fichier signe.mli

```
type t = Neg | Zer | Pos
val f : int -> t
```

Version « équivalente » en un seul fichier

```
module type Signe = sig
  type t = Neg | Zer | Pos
  val f : int -> t
end
module Signe : Signe = struct
  type t = Neg | Zer | Pos
  let f x = if x < 0 then Neg else if x = 0 then Zer else Pos
end
let reduit x = match Signe.f x with
  | Signe.Neg -> -1 | Signe.Zer -> 0 | Signe.Pos -> 1
```

Interface : .mli

fichier.ml

(le contenu de struct ... end)

type euros = int

type carte

= Number of int | Color

let p1 = fun (x,y) -> x

fichier.mli

(le contenu de sig ... end)

type carte

→ = Number of int | Color

→ val p1 : int * 'b -> int

Le .mli est compilé en .cmi

Interface : .mli

fichier.ml

(le contenu de struct ... end)

type euros = int

type carte

= Number of int | Color

let p1 = fun (x,y) -> x

fichier.mli

(le contenu de sig ... end)

type carte

= Number of int | Color

→ val p1 : int * 'b -> int

Le .mli est compilé en .cmi

Astuce : ocamlc -i file.ml génère le .mli

Pourquoi n'est-ce pas automatisé ?

Interface : .mli

fichier.ml

(le contenu de struct ... end)

type euros = int

type carte

= Number of int | Color

let p1 = fun (x,y) -> x

fichier.mli

(le contenu de sig ... end)

type carte

→ = Number of int | Color

→ val p1 : int * 'b -> int

Le .mli est compilé en .cmi

Astuce : ocamlc -i file.ml génère le .mli

Pourquoi n'est-ce pas automatisé ?

- ▶ Ça l'est !
- ▶ Mais le .mli généré donne accès à tout le module

Compilation séparée

Compilation des .ml et .mli avec “ocamlc -c” par ordre de dépendance

- ▶ **file.cmi** (compiled interface) : signature du module contenant les types et le typage des fonctions sans leur implémentation
- ▶ **puis file.cmo** (object bytecode) : représentation intermédiaire, non exécutable

Compilation séparée

Compilation des `.ml` et `.mli` avec “`ocamlc -c`” par ordre de dépendance

- ▶ **file.cmi** (compiled interface) : signature du module contenant les types et le typage des fonctions sans leur implémentation
- ▶ puis **file.cmo** (object bytecode) : représentation intermédiaire, non exécutable

Liaison des `.cmo` avec `ocamlc`

```
$ ocamlc types.cmo fifo.cmo main.cmo -o executable  
(.cmo cités dans un ordre compatible avec les dépendances)
```

Les utilisations croisées de modules sont interdites !

Compilation séparée : Exemple *

Fichiers .ml et .mli

- ▶ `geometrie.mli`, `geometrie.ml`
- ▶ `affichage.mli`, `affichage.ml` (*utilise la bibliothèque Graphics*)
- ▶ `demo.ml` (*utilise les modules Geometrie et Affichage*)

Compilation séparée : Exemple *

Fichiers .ml et .mli

- ▶ `geometrie.mli`, `geometrie.ml`
- ▶ `affichage.mli`, `affichage.ml` (*utilise la bibliothèque Graphics*)
- ▶ `demo.ml` (*utilise les modules Geometrie et Affichage*)

Génération des .cmi et .cmo

```
ocamlc -c geometrie.mli      → geometrie.cmi  
ocamlc -c geometrie.ml      → geometrie.cmo  
ocamlc -c affichage.mli     → affichage.cmi  
ocamlc -c affichage.ml      → affichage.cmo  
ocamlc -c demo.ml           → demo.cmo
```

Compilation séparée : Exemple *

Fichiers .mli et .ml

- ▶ `geometrie.mli`, `geometrie.ml`
- ▶ `affichage.mli`, `affichage.ml` (*utilise la bibliothèque Graphics*)
- ▶ `demo.ml` (*utilise les modules Geometrie et Affichage*)

Génération des .cmi et .cmo

```
ocamlc -c geometrie.mli      → geometrie.cmi  
ocamlc -c geometrie.ml      → geometrie.cmo  
ocamlc -c affichage.mli     → affichage.cmi  
ocamlc -c affichage.ml      → affichage.cmo  
ocamlc -c demo.ml           → demo.cmo
```

Liaison

```
ocamlc graphics.cma geometrie.cmo affichage.cmo demo.cmo -o demo
```

Modules et interpréteur interactif

- ▶ `#use "toto.ml";;`
lit, interprète et exécute `toto.ml`
(comme s'il était **tapé dans l'interpréteur**, pas de module)
- ▶ `#load "toto.cmo";;` ou `#load "toto.cma";;`
charge en mémoire un fichier **déjà compilé**
 - ▶ `cmo` : un fichier objet = un module
 - ▶ `cma` : bibliothèque = collection de modules
- ▶ À ne pas confondre avec `open Toto;`
qui rend accessibles les composants d'un module par leur nom court.

Plan

Modules

Motivations

En pratique

Compilation

Principes de base

Compilation séparée

Makefile

Foncteurs

Motivations

- ▶ Si je modifie UN de mes fichiers, par le jeu des dépendances je peux en avoir plusieurs à recompiler : fastidieux
- ▶ Lesquels ?
(besoin de recenser les dépendances)
- ▶ Les commandes de compilations peuvent être complexes
(options)

`make` permet d'automatiser tout cela !

Motivations

- ▶ Si je modifie UN de mes fichiers, par le jeu des dépendances je peux en avoir plusieurs à recompiler : fastidieux
- ▶ Lesquels ?
(besoin de recenser les dépendances)
- ▶ Les commandes de compilations peuvent être complexes
(options)

make permet d'automatiser tout cela !

commande

make

Utilise par défaut un fichier `Makefile`.

Exemple naïf

A la main avant

```
ocamlc -c a.ml  
ocamlc -c b.ml  
ocamlc camlp4o.cma a.cmo b.cmo -o monProg
```

Exemple naïf

A la main avant

```
ocamlc -c a.ml  
ocamlc -c b.ml  
ocamlc camlp4o.cma a.cmo b.cmo -o monProg
```

Makefile associé

```
all:  
    ocamlc -c a.ml  
    ocamlc -c b.ml  
    ocamlc camlp4o.cma a.cmo b.cmo -o monProg
```

Exemple naïf

A la main avant

```
ocamlc -c a.ml  
ocamlc -c b.ml  
ocamlc camlp4o.cma a.cmo b.cmo -o monProg
```

Makefile associé

```
all:  
    ocamlc -c a.ml  
    ocamlc -c b.ml  
    ocamlc camlp4o.cma a.cmo b.cmo -o monProg
```

Le processus est automatisé mais **on recompile tout à chaque fois.**

Structure du Makefile

- ▶ cible : objet à fabriquer
- ▶ dépendances : prérequis pour générer la cible
- ▶ commande : processus de fabrication

Code

```
cible1 : dependances1
        commande1

cible2 : dependances2
        commande2
        ...
```

Commande précédée d'une **tabulation**.

Que fait `make` `cible` ?

Évaluation

- ▶ on recherche chaque dépendance de `cible` parmi les cibles et si on la trouve, on l'évalue récursivement.
- ▶ si `cible` est inexistante ou si une de ses dépendances est plus récente que `cible`, alors la commande de compilation de `cible` est exécutée.

Par défaut `make` génère la première cible du `Makefile` (donc en général on place l'exécutable au début).

Exemple

```
all: monProg
```

```
monProg: a.cmo b.cmo
```

```
    ocamlc camlp4o.cma a.cmo b.cmo -o monProg
```

```
a.cmo: a.ml
```

```
    ocamlc -c a.ml
```

```
b.cmo: b.ml
```

```
    ocamlc -c b.ml
```

Exemple

```
all: monProg
```

```
monProg: a.cmo b.cmo
```

```
    ocamlc camlp4o.cma a.cmo b.cmo -o monProg
```

```
a.cmo: a.ml
```

```
    ocamlc -c a.ml
```

```
b.cmo: b.ml
```

```
    ocamlc -c b.ml
```

Possibilité de définir des variables `MAVAR=...` rappelées plus tard par `$(MAVAR)`

Exemple

```
EXEC=monProg
```

```
all: $(EXEC)
```

```
$(EXEC): a.cmo b.cmo
```

```
    ocamlc camlp4o.cma a.cmo b.cmo -o $(EXEC)
```

```
a.cmo: a.ml
```

```
    ocamlc -c a.ml
```

```
b.cmo: b.ml
```

```
    ocamlc -c b.ml
```

Possibilité de définir des variables MAVAR=... rappelées plus tard par \$(MAVAR)

Exemple

```
EXEC=monProg
```

```
all: $(EXEC) clean
```

```
$(EXEC): a.cmo b.cmo
```

```
    ocamlc camlp4o.cma a.cmo b.cmo -o $(EXEC)
```

```
a.cmo: a.ml
```

```
    ocamlc -c a.ml
```

```
b.cmo: b.ml
```

```
    ocamlc -c b.ml
```

```
clean:
```

```
    rm -rf *.cmi *.cmo *~
```

Possibilité de définir des variables MAVAR=... rappelées plus tard par \$(MAVAR)

Plan

Modules

- Motivations

- En pratique

Compilation

- Principes de base

- Compilation séparée

- Makefile

Foncteurs

Modularité avancée : foncteurs

- ▶ Un foncteur est un module paramétré par un autre module, autrement dit une fonction qui à “tout” module en associe un autre.
- ▶ Déclaration :

```
module Nom =  
  functor (Param : SIGNATURE) ->  
  struct ... end
```

Modularité avancée : foncteurs

- ▶ Un foncteur est un module paramétré par un autre module, autrement dit une fonction qui à “tout” module en associe un autre.

- ▶ Déclaration :

```
module Nom =  
  functor (Param : SIGNATURE) ->  
  struct ... end
```

- ▶ Application :

```
module Resultat = Nomdufoncteur(Nomdumodule)
```

- ▶ Le module donné en argument doit fournir
 - ▶ des types compatibles avec ceux de la SIGNATURE demandée
 - ▶ *au moins* les fonctions de la SIGNATURE

Exemple simple de foncteur

Module `Tri` : diverses fonctions sur un ensemble totalement ordonné

Signature du paramètre

```
module type ORDONNE =  
  sig  
    type t  
    val comp : t -> t -> int  
    (* Ordre total sur t, renvoie  
       un négatif pour inférieur,  
       0 pour égal,  
       un positif pour supérieur *)  
  end
```

Exemple simple de foncteur *

Définition du foncteur

```
module Tri =  
  functor (E : ORDONNE) ->  
    struct  
      let infegal x y = (E.comp x y <= 0)  
      let max x y = if (E.comp x y < 0) then y else x  
    end
```

Application du foncteur

```
module Entier =  
  struct  
    type t = int  
    let comp x y = x - y  
  end  
  
module TriEntier = Tri(Entier);;  
  
TriEntier.infegal 1 2;;  
- : bool = true  
  
TriEntier.max 12 2;;  
- : Entier.t = 12
```


Cacher une partie du foncteur

Signature d'un foncteur

```
module type TRI = functor (E : ORDONNE) ->  
  sig  
    val infegal : E.t -> E.t -> bool  
    val max : E.t -> E.t -> E.t  
    val tri : E.t list -> E.t list  
  end
```

```
module Tri : TRI = functor (E : ORDONNE) ->  
  struct ...  
    let rec separe x l = ...  
  
    let rec tri l = ...  
  end
```

Attention : trop d'abstraction gêne le typage

```
module type ENS =  
sig  
  type elt  
  type ens  
  ...  
end ;;
```

Attention : trop d'abstraction gêne le typage

```
module type ENS =  
  sig  
    type elt  
    type ens  
    ...  
  end ;;  
  
module Ensemble : functor (T : ORDONNE) -> ENS =  
  functor (T : ORDONNE) ->  
  struct  
    type elt = T.t  
    type ens = elt list  
    ...  
  end ;;
```

Attention : trop d'abstraction gêne le typage

```

module type ENS =
sig
  type elt
  type ens
  ...
end ;;

module Ensemble : functor (T : ORDONNE) -> ENS =
  functor (T : ORDONNE) ->
struct
  type elt = T.t
  type ens = elt list
  ...
end ;;

module EnsEntier = Ensemble(Entier);;
EnsEntier.singleton 1;;

```

Error: This expression has type int but an expression was expected of type EnsEntier.elt

Solution : signature avec contrainte

```
module Ensemble :  
  functor (T : ORDONNE) -> ENS with type elt = T.t =  
  functor (T : ORDONNE) ->  
struct  
  type elt = T.t  
  type ens = elt list  
  ...  
end ;;
```

En bref

Aujourd'hui

- ▶ Les **modules** permettent d'**organiser** et/ou d'**isoler** du code
- ▶ Un source OCaml peut être **compilé** en bytecode ou en natif
- ▶ Un source OCaml peut être réparti en plusieurs fichiers (un module par fichier), **compilés séparément** éventuellement à l'aide d'un **Makefile**
- ▶ Les **foncteurs** sont des modules **paramétrés** qui permettent une vraie programmation modulaire.