Programmation fonctionnelle avec OCaml

3ème séance, 19 mars 2015

modules & compilation

samuel.hornus@inria.fr

http ://www.loria.fr/~shornus/ocaml/

Modules

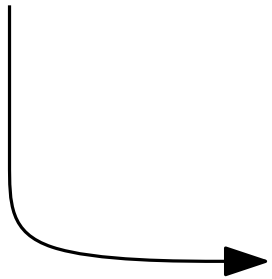
Un `module` regroupe un ensemble de définitions apparentées

- de valeurs (`let`)
- de types (`type`)
- d'exceptions (`exception`)
- de sous-modules (`module`)
- etc...

Modules

Un `module` regroupe un ensemble de définitions apparentées

- de valeurs (`let`)
- de types (`type`)
- d'exceptions (`exception`)
- de sous-modules (`module`)
- etc...



On a donc une hiérarchie de modules

Modules

Un `module` regroupe un ensemble de définitions apparentées

- de valeurs (`let`)
- de types (`type`)
- d'exceptions (`exception`)
- de sous-modules (`module`)
- etc...

Exemple : le module `List` regroupe des fonctions relatives à la manipulation des listes.
Le module `String` regroupe des fonctions relatives à la manipulation des chaînes de caractères.

Modules

Accès au contenu d'un module par la notation pointée :

```
# String.sub "fichier.ml" 8 2;;
- : string = "ml"

# List.partition (fun x -> x mod 2 = 0)
[1;2;3;4;5;6;7;8;9;10];;
- : int list * int list = ([2; 4; 6; 8; 10], [1; 3; 5; 7; 9])
```

Les modules, concrètement

Un fichier source `myMod.ml` = Un module `MyMode`

`myMod``.cmo` (*bytecode object file*) ou
`myMod``.cmx` & *`myMod`*`.o` (*native object file*)

Les modules, concrètement

Un fichier source `myMod.ml` = Un module `MyMode`

`myMod``.cmo` (*bytecode object file*) ou
`myMod``.cmx` & *`myMod`*`.o` (*native object file*)

Dans le REPL, on peut charger un module avec la directive
`#load "`*`myMod`*`.cmo"`

Les modules de la « biliothèque standard » sont chargés automatiquement lors de leur utilisation : `String`, `List`, `Array`, `Stream`, etc...

Les modules, dans le language

On peut ouvrir un module avec la directive

`open` *`Module`*

...importe les définitions du module dans l'environement courant. *La notation pointée devient inutile.*

Le module `Pervasives`[a] est ouvert automatiquement :

`int_of_float` au lieu de `Pervasives.int_of_float`

Les modules de la *"standard library"* ne sont pas ouvert automatiquement : il faut donc utiliser la notation "pointée" :

```
# String.make 4 'r';;
- : string = "rrrr"
```

---

a. *"pervasive"* = omniprésent

Les modules, dans le language

Ouverture d'un module

```
String.length "Petit patapon";;
- : int = 13
String.make 5 'O';;
- : string = "OOOOO"

open String;;

length "Petit patapon";;
- : int = 13
let i5 = make 5 'i';;
val i5 : string = "iiiii"
```

Les modules, dans le language

Ouverture d'un module

```
String.length "Petit patapon";;
- : int = 13
String.make 5 'O';;
- : string = "OOOOO"

open String;;

length "Petit patapon";;
- : int = 13
let i5 = make 5 'i';;
val i5 : string = "iiiii"
```

Attention :

`List.length`

`String.length`

`Array.length`

Les modules, dans le language

Ouverture locale d'un module

(version $\geq$ 3.12)

Deux syntaxes :

```
1 String.(length "ab" + length "cde");;
- : int = 5
```

```
Module.(expression;
        ou séquence utilisant le module)
```

```
2 let open List in sort [5;4;3;2;1];;
- : int list = [1; 2; 3; 4; 5]
```

```
let open Module in expression; ou séquence
                   utilisant le module
```

Compiler un module avec `ocamlc`

`ocamlc -c`

Pour créer un module Toto :

Fichier d'implémentation `toto.ml`
Fichier d'interface `toto.mli`

`ocamlc -c toto.mli` $\Longrightarrow$ `toto.cmi`

`ocamlc -c toto.ml` $\Longrightarrow$ `toto.cmo`

Compiler un module avec `ocamlc`

`ocamlc -c`

Pour créer un module Toto :

Fichier d'implémentation `toto.ml`
Fichier d'interface `toto.mli`

`ocamlc -c toto.mli` $\Longrightarrow$ `toto.cmi`
`ocamlc -c toto.ml` $\Longrightarrow$ `toto.cmo`

Le fichier d'implémentation `toto.ml` contient une suite de définitions.

Le fichier d'interface contient le nom et le type des composants du fichier d'implémentation que l'on souhaite rendre public.

Compiler un module avec `ocamlc`

Exemple de module pour gérer un ensemble :

Fichier ensemble.ml

```
type α t = Vide
  | Noeud of α * α t * α t
let ensembleVide = Vide
let rec ajoute v t = ...
let rec cherche_min t = ...
let rec enleve v t = ...
let rec elem v t = ...
```

Fichier ensemble.mli

```
type α t
val ensembleVide : α t
val ajoute : α → α t → α t
val enleve : α → α t → α t
val elem : α → α t → bool
```

Compiler un module avec `ocamlc`

Exemple de module pour gérer un ensemble :

Fichier ensemble.ml

```
type α t = Vide
  | Noeud of α * α t * α t
let ensembleVide = Vide
let rec ajoute v t = ...
let rec cherche_min t = ...
let rec enleve v t = ...
let rec elem v t = ...
```

Fichier ensemble.mli

```
type α t
val ensembleVide : α t
val ajoute : α → α t → α t
val enleve : α → α t → α t
val elem : α → α t → bool
```

DÉMO

Compiler un module avec `ocamlc`

Exemple de module pour gérer un ensemble :

Fichier ensemble.ml

```
type α t = Vide
  | Noeud of α * α t * α t
let ensembleVide = Vide
let rec ajoute v t = ...
let rec cherche_min t = ...
let rec enleve v t = ...
let rec elem v t = ...
```

Fichier ensemble.mli

```
type α t
val ensembleVide : α t
val ajoute : α → α t → α t
val enleve : α → α t → α t
val elem : α → α t → bool
```

DÉMO

On a rendu le type `Ensemble.t` abstrait.

La fonction `cherche_min` est inaccessible en dehors du module `Ensemble`.

Compiler un module avec `ocamlc`

S'il n'y a pas de fichier d'interface `.mli` ?

$\Rightarrow$ Alors, un `.cmi` est généré à partir du `.ml` : toutes les définitions sont exportées.

Si on a une interface de module `.mli`, il faut compiler ce `.mli` avant le `.ml` pour obtenir le fichier d'interface compilée `.cmi`

$\Rightarrow$ Alors, au moment de la compilation du `.ml`, le compilateur vérifie l'accord entre les types définis dans le `.cmi` et ceux définis dans le `.ml`

Compiler un programme avec `ocamlc`

Quand un module est chargé

- les définitions (type, fonction) sont évaluées (compilées).
- les expressions (en dehors d'une définition) sont évaluées.
- DÉMO

Compiler un programme avec `ocamlc`

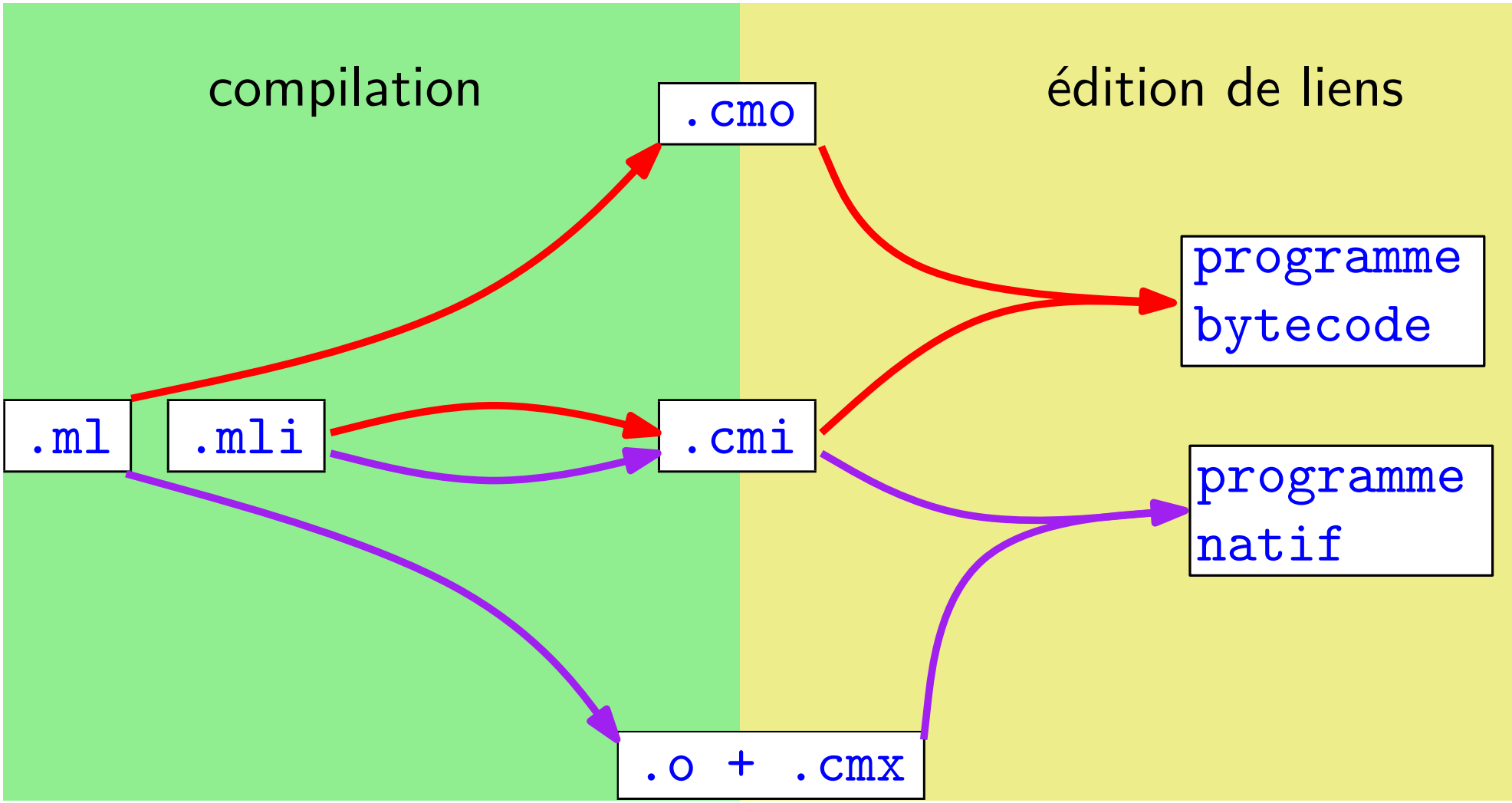
Compilation d'un programme :

`ocamlc [-o nom_prog] module_1 module_2 ... module_n`

où `module_k` ne dépend que des modules précédents `module_1`, ..., `module_k-1`

DÉMO

Compiler un programme Ocaml



`ocamlc` compilateur *bytecode* (machine virtuelle)

`ocamlopt` compilateur natif

Compiler un programme Ocaml

Quelques options pour `ocamlc` ou `ocamlopt` :

`-o` *`fichier`* indique le nom du fichier cible.

`-c` indique qu'il faut seulement compiler, pas lier.

`-g` ajoute des informations de debug.

`-I` indique un répertoire où chercher des fichiers.

`-w ...` pour la gestion des warnings.

Comme d'habitude : `man ocamlc`

Directives spéciales dans la boucle interactive `ocaml`

`#quit` ou `Control-D`	(Il faut écrire le dièse #).
`#load "fichier.cmo"`	pour charger du code compilé en mémoire.
`#use "fichier.ml"`	pour lire et évaluer le fichier.
`#trace` *`fonction`*	pour visualiser les appels à la fonction voulue.
`#untrace` *`fonction`*	

Directives spéciales dans la boucle interactive `ocaml`

`#quit` ou `Control-D`	(Il faut écrire le dièse #).
`#load "fichier.cmo"`	pour charger du code compilé en mémoire.
`#use "fichier.ml"`	pour lire et évaluer le fichier.
`#trace` *`fonction`*	pour visualiser les appels à la fonction voulue.
`#untrace` *`fonction`*	

Pour faire un script exécutable : ajouter la ligne

`#!`*`/path/to`*`/bin/ocamlrun` *`/path/to`*`/bin/ocaml`

Puis `chmod u+x` *`fichier`*

DÉMO

Interlude sur les références

Une référence permet de simuler des variables mutable.

Une valeur de type α `ref` est une « boîte » contenant une valeur de type α :

```
# let toto = ref 22;;  (* on crée un boîte *)
# !toto;;              (* on ouvre la boîte *)
- : int = 22
# toto := !toto + 13;; (* on change le contenu de
# !toto;;               la boîte *)
- : int = 35
```

Créer un module explicitement

```
# module Pile =
  struct
    type 'a t = 'a list ref
    let create () = ref []
    let push x p = p := x::!p
    let pop p = match !p with
        | [] -> failwith "Pile vide"
        | x::r -> (p := r; x)
  end;;
```

Créer un module explicitement

```
# module Pile =
  struct
    type 'a t = 'a list ref
    let create () = ref []
    let push x p = p := x::!p
    let pop p = match !p with
        | [] -> failwith "Pile vide"
        | x::r -> (p := r; x)
  end;;
module Pile :
  sig
    type 'a t = 'a list ref
    val create : unit -> 'a list ref
    val push : 'a -> 'a list ref -> unit
    val pop : 'a list ref -> 'a
  end
```

Créer un module explicitement

```
# let p = Pile.create ();;
val p : '_a list ref = contents = []

# List.iter (fun x -> Pile.push x p) ["piment";
"chou"; "radis"];;
- : unit = ()
# Pile.pop p;;
- : string = "radis"
# Pile.push "oseille" p;;
- : unit = ()
# Pile.pop p;;
- : string = "oseille"
# Pile.pop p;;
- : string = "chou"
```

Créer un module explicitement

Petit souci :

```
# !p;;
- : string list = ["piment"]
# p := !p @ ["camembert"; "quiche"];;
- : unit = ()
# Pile.pop p;;
- : string = "piment"
# Pile.pop p;;
- : string = "camembert"
```

Créer un module explicitement

Petit souci :

```
# !p;;
- : string list = ["piment"]
# p := !p @ ["camembert"; "quiche"];;
- : unit = ()
# Pile.pop p;;
- : string = "piment"
# Pile.pop p;;
- : string = "camembert"
```

On a modifié la pile `p` sans respecter le fonctionnement d'une pile. C'est pas bien ! Comment empêcher cela ?

Créer un module explicitement

Petit souci :

```
# !p;;
- : string list = ["piment"]
# p := !p @ ["camembert"; "quiche"];;
- : unit = ()
# Pile.pop p;;
- : string = "piment"
# Pile.pop p;;
- : string = "camembert"
```

On a modifié la pile `p` sans respecter le fonctionnement d'une pile. C'est pas bien ! Comment empêcher cela ?

Rappel : dans un fichier d'interface `.mli`, on peut abstraire un type, pour le rendre inaccessible.

Définir un signature

```
# module type PILE =
  sig
    type 'a t
    val create : unit -> 'a t
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
  end;;
```

Définir un signature

```
# module type PILE =
  sig
    type 'a t
    val create : unit -> 'a t
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
  end;;
```

On a abstrait le type utilisé pour représenter la pile.

Définir un signature

```
# module type PILE =
  sig
    type 'a t
    val create : unit -> 'a t
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
  end;;
```

On a abstrait le type utilisé pour représenter la pile.

Création d'un nouveau module avec la signature `PILE` :

```
# module OKPile = (Pile : PILE);;
module OKPile : PILE
```

Expliciter la signature d'un module

```
# open OKPile;;

# let p = create();;
val p : '_a OKPile.t = <abstr>
# List.iter (fun x -> push x p) ["piment"; "chou";
"radis"];;
- : unit = ()
# !p;;
Error: This expression has type string OKPile.t but an
expression was expected of type 'a ref
# pop p;;
- : string = "radis"
# pop p;;
- : string = "chou"
```

Revue du cours

On compile un fichier `.ml` pour obtenir un module `.cmo`/`.cmx`.

Revue du cours

On compile un fichier `.ml` pour obtenir un module `.cmo`/`.cmx`.

On lui adjoint souvent un fichier interface `.mli` pour :

Revue du cours

On compile un fichier `.ml` pour obtenir un module `.cmo`/`.cmx`.

On lui adjoint souvent un fichier interface `.mli` pour :

- Abstraire un type : `type` *`nom_du_type`*

Revue du cours

On compile un fichier `.ml` pour obtenir un module `.cmo`/`.cmx`.

On lui adjoint souvent un fichier interface `.mli` pour :

- Abstraire un type : `type` *`nom_du_type`*
- Cacher certaines définitions (ex : fonctions internes)

Revue du cours

On compile un fichier `.ml` pour obtenir un module `.cmo`/`.cmx`.

On lui adjoint souvent un fichier interface `.mli` pour :

- Abstraire un type : `type` *`nom_du_type`*
- Cacher certaines définitions (ex : fonctions internes)

L'ordre des arguments de `ocamlc` est important (dépendance des modules)

Revue du cours

On compile un fichier `.ml` pour obtenir un module `.cmo`/`.cmx`.

On lui adjoint souvent un fichier interface `.mli` pour :

- Abstraire un type : `type` *`nom_du_type`*
- Cacher certaines définitions (ex : fonctions internes)

L'ordre des arguments de `ocamlc` est important (dépendance des modules)

Soit on utilise la notation pointée : *`Module.nom`* ; soit on importe les définitions du module : `open` *`Module`*.

Revue du cours

On compile un fichier `.ml` pour obtenir un module `.cmo`/`.cmx`.

On lui adjoint souvent un fichier interface `.mli` pour :

- Abstraire un type : `type` *`nom_du_type`*
- Cacher certaines définitions (ex : fonctions internes)

L'ordre des arguments de `ocamlc` est important (dépendance des modules)

Soit on utilise la notation pointée : *`Module.nom`* ; soit on importe les définitions du module : `open` *`Module`*.

Un *langage de module* pour définir des modules est disponible. On approfondira cet aspect de Ocaml plus tard.