Programmation fonctionnelle

ensIIE

Semestre 3 – 2026/27

Institut Mines-Télécom
ensIIE école associée
université PARIS-SACLAY

ensIIE

www.ensiie.fr

Modularité

Modularité

GCC : $\sim$4 millions de lignes de code
Noyau Linux : $\sim$12 millions de lignes de code

- besoin de partage des tâches entre développeurs

On sépare les différentes composantes d'un logiciel dans des modules

Chaque module possède une interface qui indique quelles sont les fonctionnalités qu'il propose

En dehors du module, seul ce qui est déclaré dans l'interface peut être utilisé (boîte noire)

Interface

Contrat qui indique les fonctionnalités fournies par un module :

- définition de types (éventuellement abstraits)
- prototypes de fonctions

Les modules extérieurs utilisent ces types et ces fonctions, et uniquement ceux-là, pour pouvoir utiliser le module.

Compilation séparée

Chaque module peut être compilé indépendamment des autres modules

- puisque seule leurs interfaces l'intéresse
- il n'est donc pas nécessaire de tout recompiler à chaque modification
- ni d'attendre qu'un module soit complètement implémenté pour compiler un autre

Une fois chaque module compilé, l'éditeur de liens (*linker*) se charge de « coller » les morceaux les uns avec les autres.

Éventuellement, l'éditeur de lien peut combiner des programmes écrits dans différents langages

Interfaces en OCaml

Pour un module `truc.ml` l'interface est définie dans un fichier `truc.mli` compilée ensuite en `truc.cmi`. Elle contient :

- des définitions de type (éventuellement abstraits)

```
type s = A | B of int
type t
```

- des déclarations de type de valeurs (dont des fonctions)

```
val init : s
val bidule : char -> t -> int
```

L'implémentation `truc.ml` doit déclarer des types concrets et des valeurs dont le type est plus général que celui de l'interface.

Un autre module qui a besoin de `truc` peut soit donner explicitement le nom du module (ex : `Truc.bidule`) ou soit ouvrir le module avant utilisation (`open Truc`).

Types privés

Possibilité de définir des types semi-abstraits :

```
type t = private A | B of ...
```

ou

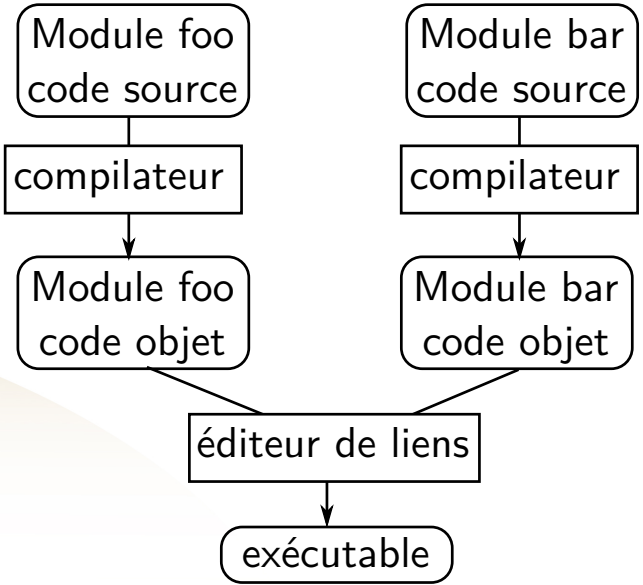
```
type t = private { x : int; y : ... }
```

On ne peut pas créer directement une valeur à l'aide des constructeurs / des enregistrements

mais on peut faire du filtrage de motif et accèder aux champs

Permet de maintenir un invariant sur la structure

Chaîne de compilation



Chaîne de compilation : OCaml

- source `foo.ml` → code objet `foo.cmo` (bytecode)

```
ocamlc -c foo.ml
```

- code objets `foo.cmo bar.cmo` → exécutable `prog`

```
ocamlc -o prog foo.cmo bar.cmo
```

Toutes les phrases des modules sont exécutées, l'ordre des modules lors de l'édition de lien est important

Il existe également pour la plupart des architectures une compilation vers du code natif, plus performant :

```
ocamlopt -c foo.ml
ocamlopt -o prog foo.cmx bar.cmx
```

Avantages de la modularité

Permet :

- séparation des tâches
- création d'un espace de nom
- réutilisabilité (par exemple, bibliothèques)
- encapsulation
- abstraction (en particulier des types)
- maintenabilité

Aparté : surcharge VS polymorphisme

- surcharge : même nom de fonction pour deux algorithmes différents

  Exemple : + en C sur les `int` et les `double`

- polymorphisme : fonction générique qui peut s'appliquer à n'importe quel type de données, même algo quelque soit le type

  Exemple : `fun x -> x` en OCaml

Extensions

En OCaml, modules plus avancés :

- modules imbriqués
- modules paramétrés par des modules (foncteurs)
  - généricité
- modules de première classe : peuvent être manipulés comme des valeurs du langage

Définition de module

```
module M = struct
  ...
end
```

À l'intérieur :

- définition de type `type t = ...`
- définition de valeurs `let f = ...`
- définition d'exception `exception E`
- définition de modules `module N = ...`

Comme dans un `.ml`

Défition d'une interface de module

```
module type S = sig
  ...
end
```

À l'intérieur :

- déclarations de type (concrets ou abstraits) `type t = ...`
- déclarations de valeurs `val f : ...`
- déclaration d'exceptions `exception E`
- déclarations de modules (concrets ou abstraits) `module N : S = ...`

Comme dans un `.mli`

Lier un module à une interface

```
module type S = sig
  val f : int -> int
end

module M : S = struct
  let a = 42
  let f x = x + a
end
```

En dehors de `M` seul f est visible.

Un `.ml` est automatiquement typé par son `.mli`

Ensembles

Vue abstraite des ensembles

Structure dynamique
Fonctions :

- ensemble vide + test de vacuité
- appartenance
- ajout
- cardinal
- union, intersection
- retrait
- agrégation, map
- égalité
- ...

Interface

```
type 'a set

val empty : 'a set

val is_empty : 'a set -> bool

val mem : 'a -> 'a set -> bool

val add : 'a -> 'a set -> 'a set

val union : 'a set -> 'a set -> 'a set

val inter : 'a set -> 'a set -> 'a set
```

Interface (cont.)

```
val remove : 'a -> 'a set -> 'a set

val equal : 'a set -> 'a set -> bool

val map : ('a -> 'b) -> 'a set -> 'b set

val fold : ('a -> 'b -> 'b) -> 'a set -> 'b -> 'b
```

Implémentation à l'aide de liste

- vide : liste vide $\mathcal{O}(1)$
- appartenance : recherche dans liste $\mathcal{O}(n)$
- ajout : ajout sans doublon $\mathcal{O}(n)$
- cardinal : longueur de liste $\mathcal{O}(n)$
- union : concaténation sans doublon $\mathcal{O}(n_1 \times n_2)$
- intersection $\mathcal{O}(n_1 \times n_2)$
- retrait : parcours $\mathcal{O}(n)$
- agrégation $\mathcal{O}(n)$, map $\mathcal{O}(n^2)$
- égalité $\mathcal{O}(n_1 \times n_2)$