

Unités de compilation : motivations

Un principe de base du génie logiciel est le découpage d'une application en plusieurs parties indépendantes appelées unités de compilation.

Pour:

- maîtriser la complexité de logiciels de grande taille, horizontalement (divide and conquer) et verticalement (niveaux d'abstractions).
- développer en équipe.
- modifier un morceau indépendamment des autres.
- recompiler uniquement le nécessaire.

Unités de compilation : Fichiers

En OCaml:

- 1 unité de compilation =
- 1 fichier interface + 1 fichier implémentation

Fichiers de même nom, avec différences extensions :

- ► Le fichier interface (.mli) définit les types (abstraits ou concrets) et les signatures des valeurs visibles depuis l'extérieur.
- ► Le fichier implémentation (.ml) contient les définitions (concrètes) de tous les types et de toutes les valeurs (visibles ou non) de l'unité de compilation.

Abstrait = sans implémentation dans le fichier. Concret = avec son implémentation dans le fichier.

Unités de compilation : Espaces de noms

- ▶ 1 unité de compilation toto.mli + toto.ml est désignée par Toto. La valeur v de Toto est désignée par Toto.v.
- ► La directive open Toto permet de désigner la valeur v de Toto par v.
- ➤ Si deux unités M et N contiennent v, et que l'on effectue open M puis open N, alors v désigne N.v.

Unités de compilation : Interfaces

Le fichier d'interface spécifie quels types ou valeurs sont accessibles depuis l'extérieur. Pour :

- cacher certains types ou valeurs car ils sont propres à une implémentation spécifique, ou pour faciliter la lecture.
- restreindre le type de certains composants exportés, ou la manière d'y accéder, de façon à ce qu'ils soient utilisés correctement.
- rendre abstraits certains types, pour les mêmes raisons.

Syntaxe:

► Mot-clé val pour les valeurs :

```
val f : int -> 'a -> a list
```

► Mot-clé type pour les types :

```
type t = A | B

type t (Ceci est un type abstrait)
```

Unités de compilation : Interfaces et leurs compilations

- Les fichiers d'inferface doivent être compilés via ocamlc -c fichier.mli
- ► Le fichier compilé porte l'extension .cmi
- ► Seul le fichier .cmi est finalement nécessaire pour les étapes ultérieures de compilation.

Lors de la compilation d'un fichier d'implémentation .ml

- ▶ s'il n'y a pas d'interface, un fichier .cmi est généré automatiquement : tous les types et valeurs sont exportés
- sinon, le compilateur vérifie que les types inférés depuis le .ml sont compatibles avec les types déclarés dans le .cmi.

Unités de compilation : dépendances

Pour concevoir une unité de compilation il est seulement nécessaire de connaître les interfaces des autres unités.

- ► Lorsqu'une unité M1 fait référence à une unité M2, on dit que M1 dépend de M2.
- ► L'unité M1 peut faire référence à M2 dans son interface M1.mli, soit dans son implémenation M1.ml.
- ▶ Dans un programme avec plusieurs unités de compilation, la relation « dépend de » forme un graphe de dépendances.

Le graphe de dépendances définit une ordre partiel de compilation (Pas de dépendances cycliques)

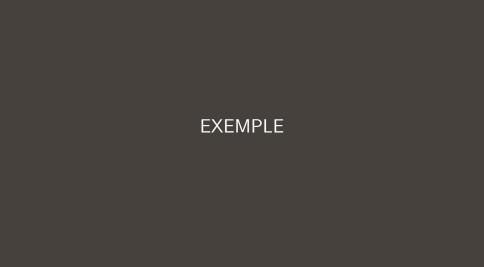
Unités de compilation : compilation vs édition de liens

► La phase de compilation effectue le typage et la production de codes à trous (on parle de fichiers objets)

```
L'option -c des compilateurs
(ocamlc ou ocamlopt pour le natif)
permet de compiler sans faire d'édition de liens
```

Les fichiers objets portent l'extension .cmo (en *bytecode*) ou .cmx (en natif).

► La phase d'édition de liens construit un exécutable en associant les trous à des implémentations—selon l'ordre des fichiers donnés en arguments.



Exemple: interface

```
Voici tri.mli :
   val tri_liste : 'a list -> 'a list
On compile :
   ocamlc -c tri.mli
Cela produit tri.cmi
```

Exemple : référence à l'interface

Un code qui utilise le module Tri dans le fichier test.ml :

```
let 11 = [4;1;5;3;2]
let 12 = Tri.tri_liste 11
```

Pour compiler, il suffit de disposer de l'interface du module Tri (par exemple dans le répertoire courant) :

```
ocamlc -c test.ml
```

Cela produit un fichier objet test.cmo.

Exemple : implémentation de l'interface

```
L'implémentation du module, dans tri.ml
  let rec insertion x l = ...
  let rec tri_liste l = ...
On compile:
  ocamlc -c tri.ml
Cela produit un fichier objet tri.cmo
```

Exemple : édition de liens

On produit un exécutable test en liant les fichiers objets test.cmo et ceux du module Tri.

```
ocamlc -o test tri.cmo test.cmo
```

On peut désormais exécuter ./test

- > ./test
- 4 1 5 3 2
- 1 2 3 4 5

Exemple : recompiler juste le nécessaire

Si on souhaite changer l'implémentation du module Tri, par exemple avec un tri rapide, il suffit de fournir une nouvelle implémentation dans un nouveau fichier tri.ml

```
let rec partage e l = ...
let rec tri_liste l = ...
```

On recompile juste ce nouveau fichier avec ocamlc -c tri.ml Inutile de recompiler test.ml.

Finalement, on refait l'édition de liens :

```
ocamlc -o test tri.cmo test.cmo
```