

# LIFPF – Programmation fonctionnelle

CM5

Compilation séparée et modules

Callbacks

Licence informatique UCBL – Printemps 2025

<https://forge.univ-lyon1.fr/programmation-fonctionnelle/lifpf/-/blob/main/README.md>

## Plusieurs fichiers source ?

Quel intérêt ?

- Se retrouver plus facilement dans le code
- Organiser le code  $\Rightarrow$  chaque morceau de code a une responsabilité limitée
- Facilite la maintenance
- Facilite le développement en équipe
- Améliore la réutilisabilité du code

# Bibliothèques

- Usage répandu dans tous les langages
- Permet de fournir un même ensemble de fonctionnalité à plusieurs programmes
- La possibilité d'avoir différents fichiers sources rend l'usage des bibliothèques possible

## Mise en œuvre : généralités

- Le code (texte ou compilé) des différents fichiers sources doit être à disposition de l'interpréteur / du compilateur
- En choisissant intelligemment un ordre de traitement, on peut gérer les fichiers sources un par un :
  - Traiter tout d'abord les fichiers qui ne **dépendent** d'aucun autre
  - Traiter ensuite les fichiers qui ne dépendent que des fichiers déjà traités
  - Itérer jusqu'à ce que tous les fichiers soient traités
- Difficultés :
  - Identifier les dépendances
  - Dépendances circulaires (à éviter)

# Mise en œuvre : interfaces et implémentations

Dans de nombreux langages (en particulier compilés) :

- On distingue :
  - une partie "publique" l'**interface**
  - une partie code et détails cachés : l'**implémentation**
- Cette séparation peut se faire en séparant l'interface et l'implémentation en deux fichiers. L'ensemble du code interface + implémentation forme une **unité de compilation**
- L'interface permet au compilateur de savoir ce qui est disponible dans une autre unité de compilation sans avoir à charger le code de cette dernière

# Interfaces et implémentation en OCaml

- Les fichiers `.ml` sont des fichiers d'implémentation
- Les fichiers d'interface sont des fichier `.mli`
  - Les fonctions y sont déclarées via la syntaxe :

```
val ma_fonction: le_type;;
```
  - Les types peuvent être définis comme dans les `.ml` ou bien on peut simplement déclarer leur existence mais pas leur structure :

```
type mon_type;;
```

- Une unité de compilation est un (cas particulier de) **module**.

# Modules en OCaml

- Les noms de module commencent par une majuscule, même si le nom de fichier commence par une minuscule.
- Pour utiliser une fonction ou un type d'un autre module, il faut la précéder du nom de ce module.
- Par exemple, la bibliothèque standard contient le module `List` qui définit entre autre la fonction `List.length`
- La déclaration

```
open MonModule;;
```

permet d'écrire `f` au lieu de `MonModule.f`

# Masquage

- L'interface peut omettre des parties de l'implémentation :
  - La structure des types
  - Certaines fonctions (qui ne sont donc pas reportées dans l'interface)
- Évite des usages non standards des modules
- Permet de définir des fonction à usage interne
- Facilite le changement d'implémentation



## Liaisons et exécutables

Dans des langages compilés

- Pour exécuter le code, on doit assembler les différentes unités de compilation. Cette phase est appelée **liaison**.
- La liaison permet de référencer correctement le code des fonctions appartenant à d'autres unités de compilation.
- Une liaison **statique** produit un exécutable intégrant le code de toutes les unités de compilation.
  - Gros exécutables
  - Facilement installables
- Une liaison **dynamique** produit un exécutable contenant juste quelques unités de compilation, voire une seule. L'exécutable contient une description des autres unités de compilation. **Au moment du lancement** le système charge en mémoire le code des autres unités de compilation et résoud les adresses de code des différents appels de fonction.

## Chaîne de compilation en OCaml

- En OCaml, la compilation va produire différents types de fichiers intermédiaires (`.cmo`, `.cmi`, `.cmx`, `.cma`, `.cmxa`, `.exe`, etc)
- Chacun de ces fichiers est produit par un appel au compilateur, au *linker*, etc
- Un outil permet d'orchestrer toute cette suite d'action : dune (il a été précédé par `ocamlbuild` et par une utilisation avancée de `make`)

# Un projet dune

- `dune-project` configuration du projet
- `bin/` fichiers liés à un exécutable
  - `xxx.ml(i)` fichiers source
  - dune configuration de l'exécutable
- `lib/` fichiers liés à une bibliothèque
  - `xxx.ml(i)` fichiers source
  - dune configuration de la bibliothèque
- `test/` fichiers liés aux tests
  - `xxx.ml(i)` fichiers source
  - dune configuration des tests

Remarque : si le projet se nomme `toto`, un module nommé `Titi` décrit par `lib/titi.ml` et `lib/titi.mli` sera accessible dans `bin` et dans `test` en le précédant de `Toto..` Par exemple on écrira `Toto.Titi.f` pour référencer la fonction `f` du module `Titi`

## Bibliothèques et opam

- Les bibliothèques peuvent être référencées dans la déclaration  
`(libraries foo bar)`  
dans un fichier dune
- L'utilitaire `opam` permet de les installer :  
`opam install une-bibliotheque`
- Leur usage se fait ensuite à travers les modules qu'elles contiennent

## Digression : le type `unit` en OCaml

- Le type des actions ( $\approx$  instructions)
- Seule valeur : `()`
- renvoyé par les fonctions ayant un effet de bord, par exemple `print_endline`

## Digression : les blocs `begin ... end`

- Contient une suite d'expressions séparées par ;
- La valeur du bloc est celle de la dernière expression
- Souvent utilisés pour enchaîner les effets de bord

### Exemple

```
begin
  print_endline "Bonjour !";
  print_endline "Je suis un programme OCaml";
  42
end
```

La valeur de ce bloc est 42

# Callbacks

- Un callback est une fonction  $c$  passée en paramètre d'une fonction  $f$
- Plutôt que de renvoyer directement une valeur,  $f$  appellera  $c$  en lui passant la valeur qu'elle a calculé
- Cette façon de faire est intéressante si  $f$  ne peut pas calculer directement le résultat ou si elle doit attendre un événement
- $c$  exprime ce qu'on fera du résultat une fois qu'il aura été obtenu

## Exemple simple : add avec callback

`add_c`

```
let add_c (x: int) (y: int) (c: int -> 'a): 'a =  
    c (x+y)  
;;
```

```
add_c 2 3 (fun v -> v);; (* Donne 5 *)
```

```
add_c 2 3 (fun v -> print_endline (string_of_int v))  
(* Affiche 5 *)
```



## Autre exemple add sur une liste

### version avec calcul au fur et à mesure

```
let rec add (init: int) (l: int list)
    (c: int -> 'a): 'a =
  match l with
  | [] -> c init
  | x::l' -> add (init+x) l' c
;;
```

### version avec mise à jour de c et calcul à la fin

```
let rec add (init: int) (l: int list)
    (c: int -> 'a): 'a =
  match l with
  | [] -> c init
  | x::l' -> add init l' (fun v -> c (x+v))
;;
```

## Live coding

Une programme pour extraire une donnée d'un fichier json