

# LIFPF – Programmation fonctionnelle

CM6

Modules, Signatures et Foncteurs

Licence informatique UCBL – Printemps 2023–2024

[https://forge.univ-lyon1.fr/programmation-fonctionnelle/lifpf/-/  
blob/main/README.md](https://forge.univ-lyon1.fr/programmation-fonctionnelle/lifpf/-/blob/main/README.md)

## <title>

### Point d'organisation

- Contrôles : 3 meilleures notes parmi 4
- Oral de TP (15/04)
- ECA (30/04 - 8h)

# Modules

- Module = regroupement de définitions
  - Jusqu'ici module = fichier `.ml`
- Une interface est une déclaration du contenu d'un module
  - Jusqu'ici interface = fichier `.mli`

Il est possible de définir un module / une interface  
à l'intérieur d'un(e) autre

En particulier dans un fichier `.ml/.mli`

## struct pour la définition de (sous-)modules

### Définition du contenu d'un module

```
struct
  (* Définitions *)
end
```

Pour définir un module, on précède la définition d'une déclaration de module :

### Définition d'un module

```
module MonModule = struct
  (* Définitions *)
end
```

# Exemple

voir code (module AssocList)

# Interfaces (ou signatures)

- Description de définitions contenues dans un module
  - Types du module
  - Valeurs (fonctions et constantes)
- Certains détails peuvent être omis
  - structure des types
  - certaines définitions

## sig pour la définition des interfaces

### Définition d'une signature

```
sig  
  (* Définitions *)  
end
```

Pour définir une interface, on précède la définition d'une déclaration :

### Définition d'une interface

```
module type MonInterface = sig  
  (* Définitions *)  
end
```

# Exemple

voir code ( $S_{\text{Assoc}}$ )



## Signatures et masquage

- On peut forcer la signature d'un module en ajoutant une référence vers une interface.
- Ce qui n'est pas dans la signature est masqué, inaccessible.

### Signature d'un module

```
module MonModule: MonInterface = struct
  ...
end
```

## Exemple

voir code (`AssocTreePb`)

Ce code est inutilisable en l'état à cause de l'inaccessibilité de  
l'implémentation du type `key`

## notation with pour exposer des implémentations de types

with après une référence d'interface

```
module MonModule: MonInterface  
  with type mon_type = un_type  
  = struct  
    ...  
  end
```

Cette notation permet d'expliciter un type d'une interface

## Exemple

voir code (`AssocTree`)

Remarque : `AssocTree` et `AssocList` sont interchangeable : on peut en effet associer la signature

`SAssoc with type key = string`

aux deux modules

## Un module qui en utilise un autre

voir code Factures

Déclaration du module Assoc comme une référence vers  
AssocList via

```
module Assoc = AssocList
```

⇒ on peut facilement changer le module Assoc en changeant  
cette déclaration.

# Foncteurs

Pour aller plus loin, on pourrait vouloir paramétrer Factures par l'implémentation choisie de Assoc :

```
module Factures
  (Assoc: SAssoc with type key = string)
  = struct
    ...
  end
```

Factures est devenu un foncteur : un module paramétré par un ou plusieurs autres modules

## Foncteurs - suite

- Les foncteurs ne peuvent pas être utilisés tels quels : il faut leur passer des modules en arguments
- Les foncteurs sont aux modules ce que les fonctions sont aux valeurs.

### Exemple : Utilisation de facture

```
module FacturesTree = Factures (AssocTree)

let _ =
  FacturesTree.string_of_facture
    (FacturesTree.ajoute "trombones"
      100 0.01 AssocTree.empty)
```

# Tour d'horizon de la bibliothèque OCaml

La bibliothèque standard d'OCaml intègre des foncteurs.

<https://v2.ocaml.org/releases/4.14/api/index.html>