

---

# Cours - TD4 : Les modules

## 1 Modules et interfaces

### 1.1 Les modules

Les modules en OCAML ont une expressivité très riche. Un module est similaire à un paquetage ADA ou à un objet JAVA (mais sans notion d'héritage ou de *self*). Voici un exemple de module :

```
module ExempleModule =  
  struct  
    include Mon_Module_Inclus  
    type mon_type = ...  
    let ma_variable = ...  
    exception Mon_exception ...  
    module Mon_Sous_Module = ...  
  end
```

La notation pointée, par exemple `ExempleModule.ma_variable`, permet l'accès aux composants d'un module, ici au composant `ma_variable`, du module `ExempleModule`. Pour faciliter cet accès en allégeant les notations, on peut "ouvrir" le module, avec `open ExempleModule`, i.e. rendre accessible directement ses constituants. Il faudra toutefois faire attention aux problèmes d'ambiguïté, dans le cas où deux modules ouverts définissent des constituants de même nom. Il est également possible d'ouvrir localement un module à l'aide de la notation pointée : `ExempleModule.(...)`. L'expression à l'intérieur des parenthèses peut être complexe (et pas seulement un composant du module) et faire référence à des variables du module.

### 1.2 Les interfaces

Un module peut se conformer à une (ou plusieurs) interfaces, similaires aux signatures de paquetages ADA ou aux interfaces JAVA. Une interface ressemble à :

```
module type ExempleInterface =  
  sig  
    include Mon_Interface_Incluse  
    type mon_type_concret = ...  
    type mon_type_abstrait  
    val ma_variable : son_type  
    exception Mon_exception ...  
    module Mon_Sous_Module : Ma_Sous_Interface  
  end
```

La déclaration (facultative) d'un module `M` respectant une interface `I` s'écrit `M : I`. Ces déclarations peuvent également être contraintes en y ajoutant des clauses définissant les types et les modules apparaissant dans l'interface, de la forme :

```
M : I with type mon_type = ... and module Mon_Sous_Module = ... and ...
```

**Remarque** : Les interfaces OCAML contenant des déclarations de types sont proches des interfaces paramétrées (par ces mêmes types) en JAVA par exemple. L'inclusion de plusieurs interfaces est possible si cela ne crée pas de conflit de symbole. Aucune identification ou unification n'est faite entre composants de mêmes noms inclus depuis plusieurs interfaces.

---

## 1.3 Lien avec les fichiers

En OCAML, un fichier `fichier.ml` (respectivement `fichier.mli`) correspond implicitement, une fois compilé, à un module `Fichier` (respectivement à une interface `Fichier`).

## 1.4 Liens module / interface

Nous avons vu que la déclaration qu'un module respecte une interface est facultative. Ces déclarations peuvent également être contraintes en y ajoutant des clauses définissant les types. Ces clauses sont facultatives et peuvent ne concerner qu'une partie des types définis dans l'interface. Nous allons étudier, sur un exemple simple les trois variantes.

<pre>module type I = sig   type t   val f : t end  module M1 = struct   type t = int   let f = 0   let g = 1 end</pre>	<pre>module M2 : I = struct   type t = int   let f = 0   let g = 1 end  module M3 : I with type t = int = struct   type t = int   let f = 0   let g = 1 end</pre>
--	---

- **M1**
  - **M1** n'est pas déclaré comme respectant l'interface **I**, tout ce qui est déclaré dans **M1** est donc visible de l'extérieur. Ainsi un appel à **M1.f**+1, depuis l'extérieur du module **M1**, s'évaluera en `- : int = 1` et l'appel à **M1.g** est autorisé.
  - Le module n'étant pas déclaré comme respectant l'interface, le compilateur ne vérifiera pas la cohérence du module et de l'interface.
  - Le module respectant l'interface, il pourra être utilisé partout où une interface **I** est attendue (voir foncteur), et ceci même si la déclaration n'a pas été faite explicitement.
- **M2**
  - **M2** est déclaré comme respectant l'interface **I**, n'est donc visible à l'extérieur de **M2** que ce qui est déclaré dans **I**. Ainsi un appel à **M2.f**+1, depuis l'extérieur du module **M2**, provoquera une erreur : `"Error : This expression has type M2.t but an expression was expected of type int"` et celui à **M2.g** `"Error : Unbound value M2.g"`.
  - Le module étant déclaré comme respectant l'interface, le compilateur vérifiera la cohérence du module et de l'interface.
  - Le module respectant l'interface, il pourra être utilisé partout où une interface **I** est attendue (voir foncteur).
- **M3**
  - **M3** est déclaré comme respectant l'interface **I** avec le type **t** étant égal à **int**, n'est donc visible à l'extérieur de **M3** que ce qui est déclaré dans **I**, ainsi que la valeur du type **t**. Ainsi un appel à **M3.f**+1, depuis l'extérieur du module **M3**, s'évaluera en `- : int = 1`. Par contre, l'appel à **M3.g** provoquera toujours l'erreur `"Error : Unbound value M2.g"`.
  - Le module étant déclaré comme respectant l'interface, le compilateur vérifiera la cohérence du module et de l'interface.
  - Le module respectant l'interface, il pourra être utilisé partout où une interface **I** est attendue (voir foncteur).

---

## 2 Utilisation des modules

### 2.1 Motivation

Lors du cours sur les parcours d'arbres, il a été précisé que : *Un parcours des éléments d'un arbre consiste à présenter ses éléments séquentiellement (i.e. "linéariser") en vue d'itérer un traitement particulier sur cette séquence. Pour simplifier le problème, on effectue une décomposition fonctionnelle en remarquant qu'un tel parcours peut se scinder en deux étapes de calcul :*

1. construire la liste (séquence) des éléments, dans l'ordre où le traitement à itérer les trouverait.
2. appliquer itérativement ce traitement sur la liste obtenue (avec un *fold* par exemple).

Nous nous étions alors intéressés uniquement à la construction de la liste. Nous avons vu que l'utilisation d'une pile permettait un parcours en profondeur de l'arbre et l'utilisation de la file un parcours en largeur.

Nous rappelons que le type des arbres binaires est le suivant :

```
type 'a standard_tree =  
  | Empty  
  | Node of 'a * 'a standard_tree * 'a standard_tree
```

Nous rappelons également le code de la construction de la liste pour un parcours en profondeur et largeur :

```
let liste_parcours_profondeur arb =  
let rec parcours pile =  
  match pile with  
  | []          -> []  
  | Empty::q    -> parcours q  
  | Node (n, g, d)::q -> n::(parcours (g::d::q))  
in parcours [arb]  
  
let liste_parcours_largeur arb =  
let rec parcours file =  
  match file with  
  | []          -> []  
  | Empty::q    -> parcours q  
  | Node (n, g, d)::q -> n::(parcours (q@[g; d]))  
in parcours [arb]
```

Si nous souhaitons trouver un élément pair de l'arbre, pour un parcours en profondeur et largeur, nous aurions le code suivant :

```
let trouver_pair_parcours_profondeur arb =  
let rec parcours pile =  
  match pile with  
  | []          -> None  
  | Empty::q    -> parcours q  
  | Node (n, g, d)::q -> if n mod 2 = 0 then Some n else (parcours (g::d::q))  
in parcours [arb]  
  
let trouver_pair_parcours_largeur arb =  
let rec parcours file =  
  match file with  
  | []          -> None  
  | Empty::q    -> parcours q  
  | Node (n, g, d)::q -> if n mod 2 = 0 then Some n else (parcours (q@[g; d]))  
in parcours [arb]
```

---

Ce TD a pour but de proposer une architecture évitant la redondance de code, en abstrayant la structure de donnée responsable du type de parcours, et le traitement à réaliser (similaire à un fold).

## 2.2 Application

### ▷ Exercice 1

1. *Écrire une interface qui abstrait les structures de données de type “collection”.*
2. *Écrire deux modules respectant cette interface, implantant une structure de pile et une structure de file.*

### ▷ Exercice 2

1. *Écrire une interface qui abstrait les paramètre d’un itérateur `fold`.*
2. *Écrire deux modules respectant cette interface, implantant la création d’une liste d’entier et la recherche d’un nombre pair.*

## 3 Les foncteurs

### 3.1 Des modules paramétrés par des modules

On peut également définir des foncteurs, i.e. des transformateurs de modules, i.e. des modules paramétrés par d’autres modules... ou d’autres foncteurs ! Ceci est analogue aux paquetages ADA avec généricité paramétrique ou aux *component diagrams* d’UML.

En OCAML, la paramétrisation des modules n’est pas autant nécessaire qu’elle peut l’être en ADA par exemple. Ceci est dû au fait qu’on dispose déjà des types génériques (comme `'a list`) d’une part et que les éléments de tout type (sauf les fonctions) disposent de comparateurs prédéfinis d’autre part, ce qui permet d’écrire rapidement des égalités, des tris, etc, sur tout type de données.

### 3.2 Application

### ▷ Exercice 3

1. *Écrire un foncteur qui permet de parcourir un arbre binaire tout en réalisant un traitement sur ses éléments. Il sera paramétré par la structure de données (qui permettra de choisir le type de parcours) et par le traitement.*
2. *Écrire, pour chaque item ci-dessous, un module qui :*
  - (a) *Crée la liste des éléments d’un arbre avec un parcours en profondeur.*
  - (b) *Crée la liste des éléments d’un arbre avec un parcours en largeur.*
  - (c) *Recherche le premier élément pair d’un arbre avec un parcours en profondeur.*
  - (d) *Recherche le premier élément pair d’un arbre avec un parcours en largeur.*