

Modules

Les notions d'implémentations et interfaces peuvent être rendues plus fines par des constructions du langage.

Définition d'une interface (signature) I dans un programme :

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

Définition d'une implémentation (module) M ayant cette interface :

```
module M : I = struct
    let a = 42
    let b = 3
    let f x = a * x + b
end
```

Le compilateur fait alors les même opérations que si I était un fichier M.mli et M un fichier M.ml

Modules paramétrés

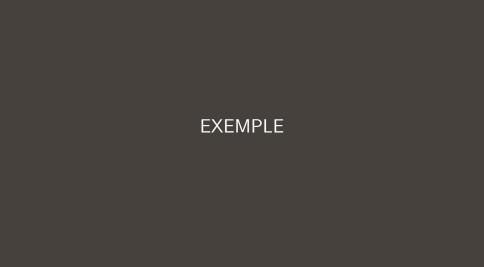
- Comme les fonctions, les modules peuvent attendre des paramètres. Ces modules paramétrés sont appelés foncteurs.
- ► Le langage impose que ces paramètres soient des modules.

Définition d'un foncteur \underline{M} qui attent un module \underline{S} de signature \underline{T} en paramètre :

```
\begin{array}{c} \text{module M (S:T) = struct} \\ & \cdots \\ \\ \text{end} \end{array}
```

Pour créer une instance de M, il suffit de l'appliquer à un module ayant la signature T. Par exemple, si B a pour signature T, alors on crée un M en faisant :

```
module A = M(B)
```



Exemple : des éléments comparables

Les éléments à trier sont rendus abstraits à l'aide d'une signature qui indique just le fait qu'ils ont une fonction de comparaison :

```
module type ELT = sig
  type t
  val compare : t -> t -> int
end
```

Le type t est abstrait.

Exemple : interface d'un trieur d'éléments comparables

Le module de tri à développer aura la signature S suivante :

```
module type S =
   sig
   type t
   val insertion : t -> t list -> t list
   val tri : t list -> t list
   end
```

Exemple : foncteur d'un trieur d'éléments comparables

Voici le foncteur correspondant :

```
module T(E : ELT) : S \text{ with type } t = E.t
 struct
  tvpe t = E.t
  type tl = t list
  let rec insertion x 1 =
    match 1 with
    | [x] <- [] |
    | v :: s ->
       if E.compare x y <= 0 then
          x :: 1
       else
           y :: (insertion x s)
  let rec tri 1 =
    match 1 with
    | [] -> []
    | x :: s -> insertion x (tri s)
end
```

Exemple : modules trieurs d'éléments comparables

```
Instanciation du foncteur T :
```

```
module E1 = struct
    type t = int
    let compare x y = y - x
  end
module T1 = T(E1)
let 11 = [4;1;5;3;-7]
let 12 = T1.tri 11
let 13 = T1.insertion 6 12
module E2 = struct
    type t = int
    let compare x y = y*y - x*x
  end
module T2 = T(E2)
let 14 = T2.tri 11
```

Exemple: préconditions...

T1.insertion prend une liste triée selon T1 comme argument, et renvoie une liste triée selon T1.

Or T1.insertion accepte toute liste de type entier. On peut toujours insérer un élément dans une liste non triée, ou mélanger les fonctions de ces deux instances :

```
let 15 = T1.insertion 4 [3;1;2] (* Bug *)
let 16 = T1.insertion 6 14 (* Bug *)
```

Il faudrait un type liste spécifique à chaque module (à chaque instance du foncteur T).

Exemple : préconditions. . . garanties par les types

Nouvelle signature pour le foncteur T avec type abstrait t1.

```
module type SL =
   sig
    type t
    type tl
   val export : tl -> t list
   val insertion : t -> tl -> tl
   val tri : t list -> tl
   end
```

Dans notre implémentation, tl sera bien une t list, mais au niveau de la déclaration de type du foncteur, rien ne le garantira : ce type tl sera donc interne à chaque instance du foncteur T.

Exemple : préconditions. . . garanties par les types

De cette manière, impossible de mélanger les listes triées par les différentes instances de T. Impossible également d'insérer un élément dans une liste non triée.

```
let 15 = T1.insertion 4 [3;1;2]
```

Error: This expression has type 'a list
but an expression was expected of type T1.tl = Sort.T(E1).tl

let 16 = T1.insertion 6 14

Error: This expression has type T2.tl = Sort.T(E2).tl
but an expression was expected of type T1.tl = Sort.T(E1).tl

Exemple: exportation type interne vers type externe

```
module T(E : ELT) : SL \text{ with type } t = E.t
    struct
     type t = E.t
     type tl = t list
     let export l = 1
   end
Pour pouvoir faire:
 List.iter (Printf.printf "%d ") (T1.export 12);
```