# Les modules en Ocaml.

Didier Rémy
2001 - 2002

http://cristal.inria.fr/~remy/mot/3/
http://www.enseignement.polytechnique.fr/profs/informatique/Didier.Remy/mot/3/

**Slide 1**

| **Cours** | **Exercices** |
|---|---|
| 1. Modules de base<br>2. Signatures<br>3. Restriction de signature<br>4. Compilation séparée<br>5. Foncteurs<br>6. Exemple avancé | 1. Abstraction tardive...<br>2. Utilisation des librairies<br>3. Création d'une librairie<br>4. Polynômes |

## En quelques mots

**Slide 2**

- Un petit langage fonctionnel typé.
- Manipule des collections de définitions (de valeurs, de types) du langage de base.
- Leurs types : des collections de déclarations/spécifications (types des valeurs, déclarations des types).
- Modules emboîtés.
- Fonctions (foncteurs) et application de fonctions.

Largement indépendant du langage de base.

## Modules de base

**Slide 3**

Les *structures* sont des collections de définitions.

`struct` $p_1 \ldots p_n$ `end`

Les définitions correspondent aux phrases du langage de base :

| | |
|---|---|
| définitions de valeurs et de fonctions | `let` $x = \ldots$ |
| définitions de types | `type` $t = \ldots$ |
| définitions d'exceptions | `exception` $E$ [`of` $\ldots$] |
| définitions de classes | `class` $C = \ldots$ `end` |

Plus :

| | |
|---|---|
| définition de sous-modules | `module` $X = \ldots$ |
| définition de type de module | `module type` $S = \ldots$ |

**Le nommage** d'un module se fait à l'aide de la liaison `module`.

```
module S = struct type t = int let x = 0 let f x = x+1 end;;
```

## Utilisation d'un module

On fait référence aux composantes d'un module avec la notation pointée *module*.*composante*

```
... (Z.f Z.x : Z.t) ...
```

Autre possibilité : la directive `open` *module* permet d'omettre le préfixe et le point :

**Slide 4**

```
open Z
... (f x : t) ...
```

## Modules emboîtés

Un module peut être composante d'un autre module :

```
module T =
  struct
    module R = struct let x = 0 end
    let y = R.x + 1
  end;;
```

**Slide 5**

La notation pointée et la construction `open` s'étendent naturellement aux sous-modules :

```
module Q =
  struct
    let z = T.R.x
    open T.R
      ...
  end
```

*NB : La notation `open T.R` rend les composantes de `T.R` visibles dans la suite du corps de `Q` mais n'ajoute pas ces composantes à `Q`.*

## Les types des modules de base

Les *signatures* : collections de spécifications (de types).

`sig` $s_1$ $s_2$ ... `end`

Slide 6

| | |
|---|---|
| spécification de valeurs | `val x :` $\sigma$ |
| spécification de types abstraits | `type` $t$ |
| spécification de types manifestes | `type` $t = \tau$ |
| spécification d'exceptions | `exception` $E$ |
| spécification de sous-modules | `module` $X : M$ |
| spécification de type de module | `module type` $S$ $[= M]$ |

Nommage d'un type de module : par la liaison `module type`

```
module type MA_SIGNATURE = sig ... end
```

## structures *v.s.* des signatures

Slide 7

| Structures (implémentations) | Signatures (interfaces) |
|---|---|
| Séquence de définitions | Séquence de spécifications de types |
| **Valeurs** | |
| `let x = [1]` | `val x : int list` |
| `let rec f x = ...` | `val f : int -> int` |
| **Exceptions** | |
| `exception E of string` | `exception E of string` |
| **Types concrets** | |
| `type t = int` | `type t = int` |
| `type 'a u = A \| B of 'a` | `type 'a u = A \| B of 'a` |
| **Types abstraits** | |
| `type v = string` | `type v` |

## Synthèse de signature

Slide 8

Le système infère les signatures des modules (comme il infère les types des valeurs).

**Module**

```
module Exemple =
  struct
    type t = int
    module R =
      struct
        let x = 0
      end
    let y = R.x + 1
  end;;
```

**Module**

**Signature inférée**

```
module Exemple :
  sig
    type t = int
    module R :
      sig
        val x : int
      end
    val y : int
  end
```

**Signature inférée**

## Restriction par une signature

Slide 9

La construction (*structure* : *signature*)

- vérifie que la structure satisfait la signature (toutes les composantes spécifiées dans la signature doivent être définies dans la structure, avec des types au moins aussi généraux) ;
- rend inaccessibles les parties de la structure qui ne sont pas spécifiées dans la signature ;
- produit un résultat qui peut être lié par `module M = ...`

Sucre syntaxique :

```
module X : S = M
```

est équivalent à

```
module X = (M : S)
```

## Restriction (Écritures équivalentes)

Slide 10

```
module M =
  (struct
     type t = int
     let x = 1
     let f y = x + y
   end :
   sig
     type t
     val f : int -> t
   end)
;;
```

```
module type S =
   sig
     type t
     val f : int -> t
   end
module M : S =
  struct
     type t = int
     let x = 1
     let f y = y + x
   end;;
```

```
M.x;;          (* Unbound value M.x *)
M.f 1 + 1;;    (* S.f 1 of type M.t is used with type int *)
```

## Vues isomorphes incompatibles

Il est parfois important de distinguer des types isomorphes. Par exemples Euros et Dollars sont tous deux représentés par des flottants. Pourtant, il ne faut pas les confondre.

Ce sont deux espaces vectoriels, isomorphes mais disjoints, avec pour unités respectives l'`euro` et le `dollar`.

Slide 11

```
module Float =
  struct
    type t = float
    let un = 1.0
    let plus = (+.)
    let prod = ( *. )
  end;;
```

```
module type MONNAIE =
  sig
    type t
    val un : t
    val plus : t -> t -> t
    val prod : float -> t -> t
  end;;
```

La multiplication devient une opération externe sur les flottants.

## Monnaies incompatibles

Dans Float le type t est concret donc il peut être confondu avec "float".
Par contre, il est abstrait dans les modules Euro et Dollar ci-dessous :

```
module Euro = (Float : MONNAIE);;
module Dollar = (Float : MONNAIE);;
```

Les types Euro.t et Dollar.t sont isomorphes mais incompatibles.

**Slide 12**

```
let euro x = Euro.prod x Euro.un;;
Euro.plus (euro 10.0) (euro 20.0);;
Euro.plus (euro 50.0) Dollar.un;;
```

Pas de duplication de code entre Euro et Dollar.

## Exercice

**Exercise 1** *On voudrait implémenter une bureau de change. Pour simplifier, nous considérons seulement deux monnaies* Euro *et* Dollar*.*
*Quel est le problème ? Avez-vous une idée de la solution ?* Answer
*Donner la signature* BUREAU_DE_CHANGE *des valeurs exportées par le module* Bureau_de_change*.* Answer
*Donner l'implémentation du module* Bureau_de_change*.* Answer
*Tester votre implémentation.* Answer □

**Slide 13**

## Vues multiples d'un même module

On peut donner une interface restreinte pour, dans un certain contexte, ne permettre que certaines opérations (typiquement, interdire la création de valeurs) :

**Slide 14**

```
module type PLUS =
  sig
    type t
    val plus : t -> t -> t
  end;;
module Plus = (Euro : PLUS)
```

```
module type PLUS_Euro =
  sig
    type t = Euro.t
    val plus : t -> t -> t
  end;;
module Plus = (Euro : PLUS_Euro)
```

À gauche, le type `Plus.t` est incompatible avec `Euro.t`.

À droite, le type `t` est partiellement abstrait et compatible avec "Euro.t", et la vue `Plus` permet de manipuler les valeurs construites avec la vue `Euro`.

## La notation `with`

La notation `with` permet d'ajouter des égalités de types dans une signature existante.

L'expression `PLUS with type t = Euro.t` est une abréviation pour la signature

**Slide 15**

```
sig
  type t = Euro.t
  val plus: t -> t -> t
end
```

On peut alors écrire

```
module Plus = (Euro : PLUS with type t = Euro.t);;
Plus.plus Euro.un Euro.un;;
```

Elle permet de créer facilement des signatures partiellement abstraites.

## Modules et compilation séparée

**Slide 16**

Une unité de compilation $A$ se compose de deux fichiers :

- Le fichier d'implémentation `a.ml` :
  une suite de phrases
  semblable à l'intérieur de `struct ... end`
- Le fichier d'interface `a.mli` (optionnel) :
  une suite de spécifications
  semblable à l'intérieur de `sig ... end`

Une autre unité de compilation `B` peut faire référence à `A` comme si c'était une structure, en utilisant la notation pointée `A.x` ou bien en faisant `open A`.

Une fois les interfaces écrites et vérifiées (par le typeur) les composantes peuvent être développées et testées indépendamment.

La modularité du développement coïncide avec la modularité dans le langage.

## Compilation séparée d'un programme

**Slide 17**

Fichiers sources : `a.ml`, `a.mli`, `b.ml`

Étapes de compilation :

| | | |
|---|---|---|
| `ocamlc -c a.mli` | compile l'interface de `A` | crée `a.cmi` |
| `ocamlc -c a.ml` | compile l'implémentation de `A` | crée `a.cmo` |
| `ocamlc -c b.ml` | compile l'implémentation de `B` | crée `b.cmo` |
| `ocamlc -o monprog a.cmo b.cmo` | édition de liens finale | |

Les lignes 2 et 3 peuvent être échangées.

Le programme se comporte comme le code monolithique :

```
module A sig (* contenu de a.mli *) end =
   struct (* contenu de a.ml *) end
module B =
   struct (* contenu de b.ml *) end
```

L'ordre des définitions de modules correspond à l'ordre des fichiers objets `.cmo` sur la ligne de commande de l'éditeur de liens.

## Utilisation de la commande `make`

**Slide 18**

## Modules paramétrés

Un *foncteur* est une fonction des modules dans les modules :

`functor` $(Z : T)$ `->` $M$

Le *module* (corps du foncteur) est explicitement paramétré par le paramètre de module $S$. Il fait référence aux composantes de son paramètre avec la notation pointée.

**Slide 19**

```
module F = functor(Z : S) ->
  struct
    type u = Z.t * Z.t
    let y = Z.f 0
  end;;
```

## Application de foncteur

On ne peut pas directement accéder dans `T`. Il faut au préalable l'appliquer explicitement à une implémentation de la signature `SIG` (tout comme on applique une fonction ordinaire).

```
module P1 = F(M1)
module P2 = F(M2)
```

**Slide 20**

`P1`, `P2` s'utilisent alors comme des structures ordinaires :

```
(P1.y : P2.u)
```

`P1` et `P2` partagent entièrement leur code.

## Exemple (long) d'un compte bancaire

**Slide 21**

```
module type BANQUE =          (* vue du banquier *)
  sig
    type t
    type monnaie
    val créer : unit -> t
    val dépôt : t -> monnaie -> monnaie
    val retrait : t -> monnaie -> monnaie
  end

module type CLIENT =          (* vue donnée au client *)
  sig
    type t
    type monnaie
    val dépôt : t -> monnaie -> monnaie
    val retrait : t -> monnaie -> monnaie
  end;;
```

## Une modélisation simple de la banque

**Slide 22**

Sur le modèle des actions, le compte est donné au client... de façon abstraite bien sûr (sa représentation est cachée) afin que seule la banque puisse modifier le compte du client.

```
module Banque_au_porteur (M : MONNAIE) :
    BANQUE with type monnaie = M.t =
  struct
    type monnaie = M.t and t = { mutable solde : monnaie }
    let zéro = M.prod 0.0 M.un and neg = M.prod (-1.0)
    let créer() = { solde = zéro }
    let dépôt c x =
      if x > zéro then c.solde <- M.plus c.solde x; c.solde
    let retrait c x =
      if c.solde > x then (c.solde <- M.plus c.solde (neg x); x) else zéro
  end;;
module Poste = Banque_au_porteur (Euro);;
```

## La modularité dans l'exemple

**Slide 23**

– Les clients et le banquier ont des vues différents, mais compatibles, de la banque.

```
module Client : CLIENT
     with type monnaie = Poste.monnaie
     with type t = Poste.t
   = Poste;;
let mon_ccp = Poste.créer ();;
Poste.dépôt mon_ccp (euro 100.0);;
Client.dépôt mon_ccp (euro 100.0);;
```

– On peut créer des comptes dans différentes devises sans risque de confusion (elles seront détectées par le typage).

```
module Citybank = Banque_au_porteur (Dollar);;
let mon_compte_aux_US = Citybank.créer();;
Citybank.dépôt mon_ccp;;
Citybank.dépôt mon_compte_aux_US (euro 100.0);;
```

## Modularité (suite)

– On peut changer l'implémentation de la banque tout en en préservant l'interface.

Pour éviter la fraude, une banque donne seulement au client un numéro de compte et conserve l'état de son compte en interne.

La représentation d'un compte devient un simple numéro.

**Slide 24**

– Plusieurs banques européennes ont alors des banques de données indépendantes (protégées).

```
module Banque_centrale = Banque_au_porteur (Euro);;
module Poste = Banque_au_porteur (Euro);;
```

**Slide 25**

```
module Banque (M : MONNAIE) : BANQUE with type monnaie = M.t =
  struct
    let zéro = M.prod 0.0 M.un and neg = M.prod (-1.0)
    type t = int
    type monnaie = M.t

    type compte = { numéro : int; mutable solde : monnaie }
    let comptes = Hashtbl.create 10 and last = ref 0
    let compte n = Hashtbl.find comptes n

    let créer() = let n = incr last; !last in
      Hashtbl.add comptes n {numéro = n; solde = zéro}; n

    let dépôt n x = let c = compte n in
      if x > zéro then c.solde <- M.plus c.solde x; c.solde
    let retrait n x = let c = compte n in
      if c.solde > x then (c.solde <- M.plus c.solde x; x) else zéro
  end;;
```

## Indépendance vis-à-vis de la représentation

`Banque_au_porteur` et "Banque" implémentent la même interface et sont interchangeables

```
module Poste = Banque_au_porteur(Euro);;
module La_Poste = Banque(Euro);;
module Citybank = Banque(Dollar);;
```

**Slide 26**

Il se trouve qu'on ne peut pas non plus observer la différence de comportement au niveau du langage, mais cela n'est pas garanti par le typage...

## Utilisation des librairies

**Exercise 2 ((*) Utilisation des librairies)** Les amis des amis sont nos amis *revient à dire que la relation d'amitié est la fermeture transitive de la relation de grande amitié (amitié directe).*

*Pour l'exercice on impose de représenter les personnes par leur nom (chaîne de caractères), les amis par des ensembles de personnes (librairie Set), et la relation de grande amitié par une table d'association (librairie Hashtbl).*

**Slide 27**

*On considère donné une certain nombre de personnes, et pour chaque personne l'ensemble de ses grands amis. Écrire une fonction qui calcule l'ensemble de tous les amis d'une personne et tester sur un petit exemple.* Answer □

## Fabrication d'un librairie

**Exercise 3 ((*) Les piles)** *Définir la structure* `'a pile` *des piles d'élément de type* `'a`*, modifiables en place . Définir une fonction de* `créer: unit -> 'a pile` *qui retourne une pile vide. Écrire les fonctions* `ajouter : 'a -> 'a pile -> unit` et `retirer : 'a pile -> 'a` *d'ajout en pile et de retrait du dernier élément. On lancera une exception lorsque l'on essaye de retirer un élément d'une pile vide.* Answer

**Slide 28**

*En faire un module* `Pile` *permettant de rendre la représentation des piles abstraite.* Answer

*Utiliser la commande make pour compiler le programme.*

*On voudrait maintenant écrire une version plus riche de* `cpile`*, avec une fonction* `consulter` *qui retourne le sommet de la pile sans le retirer, mais on ne dispose que du code compiler et de son interface.* Answer

*Que manque-t-il pour fournir une implémentation plus efficace de la méthode sommet ?*

*Modifier la librairie pile pour en avoir deux vues différentes : (1) une vue concrète, extensibles, pour l'expert ; (2) une vue abstraite pour l'utilisateur ordinaire.*

*Ici, on pourra écrire tous les modules et leur types dans un seul fichier* `piles.ml` *sans se soucier de la compilation séparée.* Answer □

**Slide 29**

## Polynômes

**Exercise 4 (Polynômes à une variable (***))**

- *Réaliser une bibliothèque fournissant les opérations sur les polynômes à une variable. Les coefficients forment un anneau passé en argument à la bibliothèque.* Answer

**Slide 30**

- *Vérifier l'égalité* $(X+Y)(X-Y) = (X^2 - Y^2)$ *en considérant les polynômes à deux variables comme polynôme en* $X$ *à coefficients les polynôme en* $Y$.
- *Vérifiez les mêmes égalités dans l'anneau Z/2Z. Z/2Z.* Answer

## Exercices (suite)

- *Écrire un programme qui prend un polynôme sur la ligne de commande et l'évalue en chacun des points lus dans* `stdin` *(un entier par ligne). Le résultat est imprimé dans* `stdout`. Answer

- *Utiliser la commande make pour compiler le programme.* Answer

**Slide 31**

- *Écrire une autre implémentation des polynômes, par exemple par des polynomes creux.*
- *Écrire une version spécialisée des polynômes dans Z/2Z utilisation une représentation pleine et des opérations logiques sur des vecteurs de bits (on pourra se limiter à des polynômes de faible degré à condition de tester le débordemnent).*

□

# 1 Solutions des exercices

## Exercise 1, page 13

Les types des monnaies Euro.t et Dollar.t sont abstraits donc incompatibles. Il n'est donc plus possibles d'écrire une fonction de coercion.

La solution est de donner dans un premier temps des versions concrète des modules Euro et Dollar, implémenter les fonctions de coercion puis d'exporter les modules Euro et Dollar de façon abstraite.

### Exercise 1 (continued)

```
module type BUREAU_DE_CHANGE =
  sig
    module Euro : MONNAIE
    module Dollar : MONNAIE
    val euro_to_dollar : Euro.t -> Dollar.t
    val dollar_to_euro : Dollar.t -> Euro.t
  end;;
```

### Exercise 1 (continued)

```
module Bureau_de_change : BUREAU_DE_CHANGE =
  struct
    module Euro = Float
    module Dollar = Float
    let un_euro_en_dollar = 0.95
    let euro_to_dollar x = x /. un_euro_en_dollar
    let dollar_to_euro x = x *. un_euro_en_dollar
  end;;
```

### Exercise 1 (continued)

```
open Bureau_de_change;;
let deux_euro = Euro.prod 2.0 Euro.un;;
let deux_euro_en_dollar = euro_to_dollar deux_euro;;
let _ = euro_to_dollar Dollar.un;;
```

## Exercise 2, page 27

amis.mlUne solution

## Exercise 3, page 28

```
type 'a pile = 'a list ref;;
let créer() = ref [];;
let ajouter x p = p := x :: !p;;
exception Vide;;
let retirer p =
  match !p with
  | [] -> raise Vide
  | x::t -> p:= t; x;;
```

**Exercise 3 (continued)**

Il suffit de placer le code précédent dans un fichier `pile.ml` et de définir l'interface suivante dans une fichier `pile.mli`.

*pile.mli*
```
type 'a pile
val créer : unit -> 'a pile
exception Vide
val ajouter : 'a -> 'a pile -> unit
val retirer : 'a pile -> 'a
```
*pile.mli*

**Exercise 3 (continued)**

On peut définit la consultation comme le retrait d'un élément suivi de son ajout ; les autres composantes sont inchangées :

*cpile.ml*
```
type 'a pile = 'a Pile.pile
let créer = Pile.créer
(* la ligne suivante, necessaire, n'est pas possible dans la version 2.04 *)

exception Vide = Pile.Vide
let ajouter = Pile.ajouter
let retirer = Pile.retirer
let consulter p = let x = Pile.retirer p in Pile.ajouter x p; x;;
```
*cpile.ml*

*cpile.mli*
```
type 'a pile
val créer : unit -> 'a pile
exception Vide
val ajouter : 'a -> 'a pile -> unit
val retirer : 'a pile -> 'a
val consulter : 'a pile -> 'a
```
*cpile.mli*

Note : Dans les versions antérieure à 3.00, le renommage d'une exception "`exception Vide = Pile.Vide`" n'était pas possible. Le seul moyen de contourner le problème est alors d'exporter une fonction permettant d'attraper une exception (un *handler*) :
```
let attrape_vide f x f' x' =
  try f x with Vide -> f' x';;
```
dans le fichier `pile.ml` avec l'interface suivante :
```
val attrape_vide : ('a -> 'b) -> 'a -> ('c -> 'b) -> 'c -> 'b
```
Et de ré-exporte `attrape_vide` dans `cpile.ml`.

**Exercise 3 (continued)**

Pour écrire une version plus efficace, il faudrait connaître la réprésentation des piles.

*piles.ml*
```
module Pre_Pile =
  struct
    type 'a pile = 'a list ref
    let créer() = ref []
    let ajouter x p = p := x :: !p
    exception Vide
    let retirer p =
      match !p with
      | [] -> raise Vide
      | x::t -> p:= t; x
  end;;
```

```
module type PILE =
  sig
    type 'a pile
    val créer : unit -> 'a pile
    exception Vide
    val ajouter : 'a -> 'a pile -> unit
    val retirer : 'a pile -> 'a
  end;;

module Pile = (Pre_Pile : PILE);;

module type CPILE =
  sig
    include PILE
    val consulter : 'a pile -> 'a
  end

module Cpile : CPILE = struct
  open Pre_Pile
  type 'a pile = 'a Pre_Pile.pile
  let créer = créer
  exception Vide = Vide
  let ajouter = ajouter
  let retirer = retirer
  let consulter p =
    match !p with
    | [] -> raise Vide
    | x::t -> x
end;;
```

*piles.ml*

### Exercise 4, page 30

*polynome.mli*

```
(* la structure d'anneau *)
module type ANNEAU =
  sig
    type t                          (* type des éléments de l'anneau *)
    val zéro : t
    val unité : t
    val nég : t -> t
    val plus : t -> t -> t
    val mult : t -> t -> t
    val equal : t -> t -> bool
    val string : t -> string
  end
      ;;

(* la structure d'anneau sur des valeurs de type t *)
module type POLYNOME =
  sig
    type c
        (* type des coefficients *)
    type t
        (* type des polynômes *)
    val zéro : t
```

```
    val unité : t
        (* unité pour le produit des polymôme.
           Superflue, car c'est un cas particulier de monôme, mais cela
           permet à la structure de polynôme d'être une superstructure de
           celles des anneaux *)
    val nég : t -> t
    val monôme : c -> int -> t
        (* monôme a k retourne le monôme a X^k *)
    val plus : t -> t -> t
    val mult : t -> t -> t
    val equal : t -> t -> bool
    val string : t -> string
    val eval : t -> c -> c
  end
      ;;


(* Le fonction qui étant donné une structure d'anneau retourne une structure
de polynôme. Il faut faire attention à réexporter le types des coefficients
de façon partiellement abstraite afin que les opérations sur les coefficients
des polynômes puissent être manipulés de l'extérieur *)


module Make (A : ANNEAU) (X : sig val nom_variable : string end)
    : (POLYNOME with type c = A.t)
    ;;
```

polynome.mli

polynome.ml

```
(* la structure d'anneau *)
module type ANNEAU =
  sig
    type t
    val zéro : t
    val unité : t
    val nég : t -> t
    val plus : t -> t -> t
    val mult : t -> t -> t
    val equal : t -> t -> bool
    val string : t -> string
  end
      ;;

(* la structure d'anneau sur des valeurs de type t *)
module type POLYNOME =
  sig
    type c
    type t
    val zéro : t
    val unité : t
    val nég : t -> t
    val monôme : c -> int -> t
    val plus : t -> t -> t
    val mult : t -> t -> t
    val equal : t -> t -> bool
    val string : t -> string
```

```
    val eval : t -> c -> c
  end
    ;;

module Make (A : ANNEAU) (X : sig val nom_variable : string end) =
  struct
    type c = A.t

    (* un monome est un coeficient et une puissance *)
    type monôme = (c * int)
    (* un polynome est une liste de monomes triés par rapport au puissance *)

    (* il est essentiel de préserver cet invaraint dans le code ci-dessous *)
    type t = monôme list

    (* pour que la représentation soit canonique, on élimine aussi
       les coeficients nuls, en particulier le monome nul est la liste vide *)
    let zéro = []

    (* on peut donc (grâce à l'invariant utiliser l'égalité générique *)
    let rec equal p1 p2 =
      match p1, p2 with
      | [],[] -> true
      | (a1, k1)::q1, (a2, k2)::q2 -> k1 = k2 && A.equal a1 a2 && equal q1 q2
      | _ -> false

    (* création d'un monôme *)
    let monôme a k =
      if k < 0 then failwith "monôme: exposant négatif"
      else if A.equal a A.zéro then [] else [a, k]

    (* un cas particulier l'unité *)
    let unité = [A.unité, 0]

    (* attention à respecter l'invariant et ordonner les monômes *)
    let rec plus u v =
      match u, v with
        (x1, k1)::r1 as p1, ((x2, k2)::r2 as p2) ->
          if k1 < k2 then
            (x1, k1):: (plus r1 p2)
          else if k1 = k2 then
            let x = A.plus x1 x2 in
            if A.equal x A.zéro then plus r1 r2
            else (A.plus x1 x2, k1):: (plus r1 r2)
          else
            (x2, k2):: (plus p1 r2)
      | [], _ -> v
      | _ , [] -> u

    (* on pourrait faire beaucoup mieux pour éviter de recalculer les
      puissances de $k$, soit directement, soit en utilisant une
      mémo-fonction *)

    let rec fois (a, k) = (* on suppose a <> zéro *)
      function
        | [] -> []
        | (a1, k1)::q ->
```

```
        let a2 = A.mult a a1 in
        if A.equal a2 A.zéro then fois (a,k) q
        else (a2, k + k1) :: fois (a,k) q

  (* on pourrai écrire : ``fois (A.nég A.unité) p'' *)
  let nég p = List.map (function a, k -> A.nég a, k) p

  let mult p = List.fold_left (fun r m -> plus r (fois m p)) zéro

  (* une impression grossière *)
  let string_monome (c,k) =
    if k = 0 then A.string c
    else if A.equal A.zéro c then "0"
    else
      let string_k =
        if k = 1 then X.nom_variable
        else String.concat "^" [ X.nom_variable; string_of_int k] in
      if A.equal A.unité c then string_k else (A.string c)^string_k
  let string p =
    if p = [] then "0"
    else "(" ^ (String.concat " + " (List.map string_monome p)) ^ ")"

  (* Puissance c^k, c un coefficient, k un entier >= 0. Par dichotomie. *)
  let rec puis c = function
    | 0 -> A.unité
    | 1 -> c
    | k ->
        let l = puis c (k lsr 1) in
        let l2 = A.mult l l in
          if k land 1 = 0 then l2 else A.mult c l2

  let eval p c = match List.rev p with
  | [] -> A.zéro
  | (h::t) ->
      let (* Réduire deux monômes en un. NB: on a k >= l. *)
          dmeu (a, k) (b, l) =
        A.plus (A.mult (puis c (k-l)) a) b, l
      in
      let a, k = List.fold_left dmeu h t in
      A.mult (puis c k) a

end
  ;;
```

*polynome.ml*

**Exercise 4 (continued)**

*polytest.ml*

```
(* pour charger le fichier compilé polynome.cmo dans le mode interactif *)

#cd "../3";;
#load "polynome.cmo";;
open Polynome;;
```

```
(* l'anneau des entiers *)
module I =
  struct
    type t = int
    let zéro = 0
    let unité = 1
    let nég x = - x
    let plus = ( + )
    let mult = ( * )
    let equal = ( = )
    let string = string_of_int
end;;

(* l'anneau des Z/2Z *)
(* invariant: on maintient la représentation canonique 0 ou 1 *)
module Z_2Z =
  struct
    type t = int
    let zéro = 0
    let unité = 1
    let plus x y = (x + y) mod 2
    let nég x = x
    let mult x y = (x * y) mod 2
    let equal = ( = )
    let string = string_of_int
end;;

(* On abstrait le test pour le répéter avec différents anneaux de base *)

module Test(A: ANNEAU) =
  struct
    (* les polynômes à coefficients dans A *)
    module X = Make (A)(struct let nom_variable = "X" end)

    (* A toplevel, poour mettre au point:
       let print_X x = Format.print_string (X.string x)
       #install_printer print_X
    *)

    (* on construit (1 + X)(1 - X) = 1 - X^2 *)
    let ( ^ ) = X.monôme
    let ( + ) = X.plus
    let ( * ) = X.mult
    let ( - ) p q = p + (X.nég q)
    let z = A.unité
    let x0, x1, x2 = z^0, z^1, z^2
    let p = (x0 + x1) * (x0 - x1)
    let q = x0 - x2
    let () =
      Printf.printf "%s %s %s\n"
        (X.string p) (if X.equal p q then "=" else "<>") (X.string q)

   (* les polynomes à coefficients dans X *)
    module Y = Make (X)(struct let nom_variable = "Y" end)

   (* Idem
      let print x = Format.print_string (Y.string x)
```

```
     #install_printer print *)

    let ( ^^ ) = Y.monôme
    let ( ++ ) = Y.plus
    let ( ** ) = Y.mult
    let ( -- ) p q = p ++ (Y.nég q)
    let x1, x2, y1, y2 = x1^^0, x2^^0, x0^^1, x0^^2
    let p = (x1 ++ y1) ** (x1 -- y1)
    let q = x2 -- y2

    let () =
      Printf.printf "%s %s %s\n"
        (Y.string p) (if Y.equal p q then "=" else "<>") (Y.string q)

  end;;


(* Le test pour I et pour Z_2Z *)
let () = print_string "\n(* Dans l'anneau des entier*)\n\n" in
let module T1 = Test(I) in
let () = print_string "\n(* Dans l'anneau Z/2Z *)\n\n" in
let module T2 = Test(Z_2Z) in
();;
```
polytest.ml

**Exercise 4 (continued)**

poly.ml
```
open Polynome

(* L'anneau des flottants --c'est plus intéressant que les entiers... *)
module F =
  struct
    type t = float
    let zéro = 0.
    let unité = 1.
    let nég x = -. x
    let plus = ( +. )
    let mult = ( *. )
    let equal = ( = )
    let string = string_of_float
  end;;
module X = Make (F) (struct let nom_variable = "X" end);;

(* construction d'un polynome: le coefficients en X^k est représenté
   par un flottant à l'indice k du tableau *)
let poly t =
  Array.fold_left
    (fun r (s, k) -> X.plus (X.monôme (float_of_string s) k) r)
    X.zéro
    (Array.mapi (fun i x -> (x, i)) t)
    ;;

(* il faut enlever le premier argument qui est le nom de la commande *)
let suffix t k = Array.sub t k (Array.length t - k);;

let main () =
```

```
  let écho, k =
    if Array.length Sys.argv > 1 && Sys.argv.(1) = "-v" then true, 2
    else false, 1 in
  let t = suffix Sys.argv k in
  let p = poly t in
  let print_eval x =
    if écho then Printf.printf "P(%f) = " x;
    print_float (X.eval p x); print_newline() in
  if écho then Printf.printf "P = %s\n" (X.string p);
  while true do print_eval (float_of_string (read_line())) done
    ;;

try main() with End_of_file -> ();;
```

*poly.ml*

*poly.sh*

```
#!/bin/sh
#
#   Exécuter les commandes sous le shell ou sauver ce fichier
#   dans poly.sh et faire sh poly.sh

./poly 1. 2. 1. <<EOF
0.5
1.
1.5
2.
2.5.
3.
EOF
```

*poly.sh*

**Exercise 4 (continued)**

Voici une solution avec le Makefile générique.