## Modules et sous-modules

Un module peut contenir des sous-modules. Le principe de la *portée lexicale* (angl. : *lexical scoping*) s'applique.

Un module contient des définitions de

1. Types (concrets ou abstraits)
2. Valeur (incl. des fonctions)
3. Exceptions
4. Modules

```
(* Demo: modules avec plusieurs interfaces et partage de type
(*****************************************************************

(* Définition d'un module IntStack *)
module IntStack =
  struct
    type stack = int list
    let empty = []
    let push l i = i::l
    let rec somme l = match l with
        [] -> 0
      | h::r -> h+(somme r)
    exception Empty_Stack
    let top l = match l with
        h::r -> h
      | [] -> raise Empty_Stack
    let pop l = match l with
        h::r -> r
```

```
      | [] -> raise Empty_Stack
  end


(* Le type stack du module IntStack est un type concret, on p
   les valeurs du type IntStack.stack comme des listes.
*)
let x = IntStack.empty;;
List.hd (IntStack.push x 42);;


(* Définition d'une signature STACK *)
module type STACK =
  sig
    type stack
    val push : stack -> int -> stack
    val empty : stack
    exception Empty_Stack
```

```
    val top : stack -> int
    val pop: stack -> stack
    val somme : stack -> int
  end


(* Définition d'un nouveau module Stack qui est la restricti
   par la signature STACK *)
module Stack = (IntStack : STACK)


(* Le type stack du module Stack est un type abstrait, on ne
   confondre avec les listes.
*)
let x = Stack.empty;;
List.hd (Stack.push x 42);;
Stack.top (Stack.push x 42);;
```

```
(* Définition d’une nouvelle signature CONSTRUCT_STACK *)
module type CONSTRUCT_STACK =
  sig
    type stack
    val empty: stack
    val push: stack -> int -> stack
  end

(* Définition d’un nouveau module CStack qui est la restricti
   par la signature CONSTRUCT_STACK *)
module CStack = (Stack : CONSTRUCT_STACK)


(* Définiton d’une nouvelle signature DESTRUCT_STACK *)
module type DESTRUCT_STACK =
  sig
    type stack
    exception Empty_Stack
```

```
    val top : stack -> int
    val pop: stack -> stack
  end


(* Définition d'un nouveau module CStack qui est la restricti
   par la signature DESTRUCT_STACK *)
module DStack = (Stack : DESTRUCT_STACK)


(* Problème: Les types stack des modules CStack et DStack son
   abstraits, l'identité de leur implantation n'est pas visib
   l'extérieur.
*)
let x = CStack.empty;;
DStack.top (CStack.push x 42);;
```

```
(* Rédéfinition des deux modules Stack et RStack, mais cette
   partage du type stack.
*)
module CStack = (Stack : CONSTRUCT_STACK with type stack = S
module DStack = (Stack :  DESTRUCT_STACK with type stack = S


(* Maintenant les deux types sont égaux mais toujours abstra
let x = CStack.empty;;
DStack.top (CStack.push x 42);;
List.hd (CStack.push x 42);;


(*************************************************************
(* Alternative: Utiliser un module local
(*************************************************************

(* Définition d'un module InstStack qui contient la définiti
```

```
   stack, puis deux modules locales qui peuvent « voir » la
   du type stack (=> principe de la Portée Lexicale).
*)

module IntStack =
  struct
    type stack = int list
    module CStack =
    struct
      let empty = []
      let push l i = i::l
    end
    module DStack =
    struct
      exception Empty_Stack
      let top l = match l with
          h::r -> h
        | [] -> raise Empty_Stack
```

```
      let pop l = match l with
          h::r -> r
        | [] -> raise Empty_Stack
    end
  end


(* Définition d'une signature qui fait le type stack abstrai
module type STACK =
sig
    type stack
    module CStack :
      sig
        val empty : stack
        val push : stack -> int -> stack
      end
    module DStack :
      sig
```

```
        exception Empty_Stack
        val top : stack -> int
        val pop : stack -> stack
      end
end


(* Restriction du module IntStack par la signature STACK *)
module Stack = (IntStack : STACK)


(* Maintenant les deux modules Stack.CStack et Stack.DStack
   le type abstrait stack.
*)
let x = Stack.CStack.empty;;
Stack.DStack.top (Stack.CStack.push x 42);;
List.hd (Stack.CStack.push x 42);;
```

```
(* Ou avec ouverture de l’espace de nom du module Stack *)
open Stack
let x = CStack.empty;;
DStack.top (CStack.push x 42);;
List.hd (CStack.push x 42);;
```

# 5 Modules paramétrés

## 5.1 Des types polymorphes au modules paramétrés

### Types polymorphes

Structure des listes polymorphes `'a list` : Les opérations sur les listes (`hd`, `tl`) sont complètement indépendantes de la structure paramètre et de ses opérations.

**Parfois les types polymorphes ne sont pas satisfaisantes**

Exemple : On veut définir un type `'a set` d'ensembles finis des éléments d'un type paramètre `'a`, mais on veut implanter les ensembles par des arbres équilibrés.

$\Rightarrow$ La réalisation des opérations sur le type `'a set` utilise l'opération de comparaison sur le type `'a`, c'est-à-dire une opération

```
compare : 'a -> 'a -> int
```

où (compare x y) renvoie
- une valeur négative quand $x$ est plus petit que $y$
- $0$ quand $x$ est égal à $y$
- une valeur positive quand $x$ est plus grand que $y$

**Solution avec un type polymorphe**

On définit un type polymorphe `'a set`, puis on passe la fonction de comparaison comme argument supplémentaire aux opérations sur les ensembles.

```
(* type polymorphe des ensembles finis sur le type 'a
type 'a set
val insert: 'a set -> 'a -> ('a -> 'a -> int) -> 'a se
val remove: 'a set -> 'a -> ('a -> 'a -> int) -> 'a se
...
```

Trop lourd, on voudrait dire une fois pour toutes quelle est l'opération de comparaison à utiliser pour un certain type paramètre.

## Solution avec un module paramétré

```
module Nom1 (Nom2: signature) = structure
```

On trouve aussi (par exemple dans la doc de OCaml :

```
module Nom1 = functor (Nom2: signature) -> structure
```

Dans la définition de `structure` on a accès aux définitions de la structure paramètre (selon la signature `signature`.

(voir la demo)

```
(* Demo: Modules paramétres.
(*****************************************************

(* Définition d'une signature pour les modules qui définent u
   ordonné.
*)

module type OrderedType =
sig
  type t
  val compare: t -> t -> int
end
;;


(* Définition d'une structure de paires d'entiers, avec ordre
   lexicographique
 *)
```

```
module OrderedIntPair =
struct
  type t = int * int
  let compare (x1,x2) (y1,y2) =
    let d1 = x1 - y1
    in if d1 = 0 then x2 - y2  else d1
end
;;


(* Définition d’un module paramétré pour les ensembles *)
module Set ( Element : OrderedType ) =
   struct
     type ele = Element.t
     type set =
 Empty
       | Node of set * ele * set
     let emptyset = Empty
```

```
    let rec insert x = function
 Empty -> Node(Empty,x,Empty)
       | Node(l,y,r) -> let c = Element.compare x y in
   if c < 0 then Node( (insert x l), y, r)
   else if c = 0 then Node( l, y, r)
   else Node( l, y, (insert x r))
    let rec member x = function
 Empty -> false
       | Node(l,y,r)  -> let c = Element.compare x y in
   if c < 0 then member x l
   else c = 0 || member x r
   end
;;


(* Application du functor Set à la structure OrdererIntPair
module IntPairSet = Set ( OrderedIntPair );;
```

```
(* Tester le module IntPairSet *)
open IntPairSet;;

let s = insert (1,0) (insert (2,4) emptyset);;
member (1,0) s;;
member (17,42) s;;


(* Pourquoi OCaml garde dans la signature de IntPairSet
   l'equation suivante?

   type set = Set(OrderedIntPair).set = Empty | Node of set

 *)


(*
   Explication: deux applications du meme foncteur a la meme
```

```
   structure donnent des types compatibles, en accord avec la
   vision fonctionnelle des foncteurs.

   Verification.
 *)

module IntPairSet2 = Set ( OrderedIntPair );;
open IntPairSet2;;

let s' = insert (1,0) (insert (2,4) s);;



(* Définition d'une signature qui laisse le type set abstrai
module type ABSET =
  sig
    type ele
```

```
    type set
    val emptyset : set
    val insert : ele -> set -> set
    val member : ele -> set -> bool
  end


(* Définition d'un functor qui, à partir d'une structure ord
   la structure d'ensemble avec un type abstrait set.
*)
module AbSet ( Element : OrderedType ) =
  ( Set ( Element ) : ABSET  with type ele = Element.t )


module BadAbSet ( Element : OrderedType ) =
  ( Set ( Element ) : ABSET)
```

```
(* Application à la structure des paires d'entiers *)
module IntPairSet = AbSet ( OrderedIntPair );;


(* Tester le nouveau module IntPairSet *)
open IntPairSet;;

let s = insert (1,0) (insert (2,4) emptyset);;
member (1,0) s;;
member (17,42) s;;

(* attention, le with type est essentiel! *)


(* Définition alternative d'un module paramétré pour les ens
(* on voit bien que la signature en sortie a le meme effet q

module Set ( Element : OrderedType ) =
```

```
(
   struct
     type ele = Element.t
     type set =
 Empty
       | Node of set * ele * set
     let emptyset = Empty
     let rec insert x = function
 Empty -> Node(Empty,x,Empty)
       | Node(l,y,r) -> let c = Element.compare x y in
   if c < 0 then Node( (insert x l), y, r)
   else if c = 0 then Node( l, y, r)
   else Node( l, y, (insert x r))
     let rec member x = function
 Empty -> false
       | Node(l,y,r)  -> let c = Element.compare x y in
   if c < 0 then member x l
   else c = 0 || member x r
```

```
   end
:   sig
    type ele = Element.t
    type set
    val emptyset : set
    val insert : ele -> set -> set
    val member : ele -> set -> bool
  end
)
;;


module IntPairSet3 = Set ( OrderedIntPair );;
open IntPairSet3;;

let s' = insert (1,0) (insert (2,4) s);;
```

```
(* enfin, encore une fonctionnalite utile quand vous souhaite
   modifier legerement une interface sans avoir a tout reecri

module type ABSET2 =
  sig
    type ele
    type set
    val emptyset : set
    val insert : ele -> set -> set
    val member : ele -> set -> bool
    val isempty : set -> bool
  end


module type ABSET2 =
  sig
    include ABSET
    val isempty : set -> bool
```

```
    val isempty : int -> bool
  end

module type ABSET3 =
  sig
    include ABSET
    with type ele = int
  end

module type Tst =
  sig
    val isempty : int -> bool
    val isempty : int -> int
    val isempty : int -> int
  end


module Tst =
```

```
struct
  let isempty x = x^"a"
  let isempty x = x+1
end
```

# 6 Encapsulation et état

L’encapsulation permet de cacher une variable dans un module, et de n’y permettre accès que par certains fonctions. Exemple : Un compteur.

### Interface

```
(* Module implementing a single counter, initialized at 0 *)

(* (inrement ()) increases the value of the counter by 1 *)
val increment: unit -> unit
(* (show ()) returns the current value of the counter *)
val show: unit -> int
```

## Corps

```
let c = ref 0
let increment () = c := !c+1
let show () = !c
```

**Exercice:** On souhaite définir un module qui permet de créer un nombre arbitraire de compteurs. L'interface est

```
(* Module defining a type of counters. Instances of counter c
   dynamically. *)

type counter = {
  increment: unit -> unit;
  show: unit -> int
}
(* create a new counter *)
val create: unit -> counter
```

Voici un exemple de l'utilisation de ce module

```
open Multicounter;;
```

```
(* this directive is only needed in the interactive toplevel
 #load "multicounter.cmo";;

let a = create ();;
a.show ();;
a.increment ();;
let b = create ();;
a.increment ();;
a.show ();;
b.increment ();;
b.show ();;
```

Réaliser le corps de ce module.