

Chapitre 5 - Encapsulation, structures fonctionnelles

INF6120: Programmation fonctionnelle et logique

Quentin Stiévenart

Université du Québec à Montréal

v251



Concept de module

Concept de module

En général, un module est un groupement de définitions

Utilité :

- **espace de nom** : on regroupe des choses sous un même nom
- **abstraction** : on cache certaines informations non nécessaires (méthode privées)
- **réutilisation de code**

Exemple de modules

En Java :

- une classe regroupe des définitions
- un *package* regroupe des classes

```
class Math {  
    public static int max(int x, int y) {  
        ...  
    }  
}
```

Exemple de modules

En Python : un fichier définit un module

```
# Fichier mymath.py
```

```
def max(n1, n2):
```

```
    ...
```

```
# Autre fichier
```

```
import mymath
```

```
mymath.max(2, 3)
```

Exemple de modules

JavaScript : plus compliqué

- pas de module initialement, état global dans une page web
- aujourd'hui, plusieurs systèmes de modules : Node, WebPack, modules ECMAScript, ...

Définition d'un module en OCaml

```
module IntPair = struct  
  type t = int * int  
  let make (x : int) (y : int) : t = (x, y)  
end
```

Un module peut contenir :

- des définitions de types
- des définitions de fonctions
- des définitions d'exceptions
- d'autres modules

Convention :

- un nom de module s'écrit en PascalCase
- un module définissant un type définit un type t

Type d'un module

```
(* La définition entrée dans UTop *)  
# module IntPair = struct  
  type t = int * int  
  let make (x : int) (y : int) : t = (x, y)  
end ;;  
  
(* Le résultat : on a le type (la signature) du module *)  
module IntPair : sig  
  type t = int * int  
  val make : int -> int -> t  
end
```

Le type d'un module est sa *spécification*

Abstraction avec les modules

On peut volontairement cacher de l'information en donnant un type au module

```
module type INT_PAIR = sig  
  type t (* définition inaccessible par l'utilisateur *)  
  val make : int -> int -> t  
end
```

```
module IntPair : INT_PAIR = struct  
  type t = int * int  
  let make (x : int) (y : int) : t = (x, y)  
end
```

Convention : un type de module s'écrit en majuscule, séparées par des _

Abstraction avec les modules

```
module Math = struct
  let rec fact_aux n acc =
    if n = 0 then
      acc
    else
      fact_aux (n - 1) (n * acc)

  let fact n = fact_aux n 1
end
```

On ne souhaite pas exposer `fact_aux` à l'utilisateur

Abstraction avec les modules

```
module type MATH = sig
  val fact : int -> int
end
```

```
module Math : MATH = struct
  let rec fact_aux n acc =
    if n = 0 then
      acc
    else
      fact_aux (n - 1) (n * acc)

  let fact n = fact_aux n 1
end
```

En Java

C'est similaire à :

```
class Math {  
    private static factAux(int n, int acc) {  
        ...  
    }  
    public static fact(int n) {  
        return factAux(n, 1);  
    }  
}
```

Abstraction avec les modules

On peut voir :

- le type du module comme ses champs *publics*
- son implémentation comme contenant des champs *privés* supplémentaires.

Fichiers et modules

Par défaut, un fichier est un module. Le fichier `foo.ml` contenant :

```
let f x = x
```

Est équivalent à :

```
module Foo = struct  
  let f x = x  
end
```

Un module pour les piles

Un module pour les piles

Une utilisation fréquente des modules et pour implémenter un type de données

```
module type STACK = sig
  exception Empty
  type 'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val push : 'a -> 'a t -> 'a t
  val peek : 'a t -> 'a
  val pop : 'a t -> 'a t
  val size : 'a t -> int
end
```

Il est fréquent d'utiliser `t` pour dénoter le type implémenté par un module

Une implémentation de STACK

```
module ListStack : STACK = struct
  exception Empty
  type 'a t = 'a list
  let empty = []
  let is_empty = function [] -> true | _ -> false
  let push x s = x :: s
  let peek = function [] -> raise Empty | x :: _ -> x
  let pop = function [] -> raise Empty | _ :: s -> s
  let size = List.length
end
```

Une autre implémentation de STACK

Implémentation différente du même type de module :

```
module ListStackCachedSize : STACK = struct
  exception Empty
  type 'a t = 'a list * int
  let empty = ([], 0)
  let is_empty = function ([], _) -> true | _ -> false
  let push x (stack, size) = (x :: stack, size + 1)
  let peek = function ([], _) -> raise Empty | (x :: _, _) -> x
  let pop = function
    | ([], _) -> raise Empty
    | (_ :: stack, size) -> (stack, size - 1)
  let size (_, size) = size
end
```

Utiliser un module

En préfixant avec le nom :

```
# ListStack.push 5 ListStack.empty;;  
- : int ListStack.t = <abstr>
```

En préfixant avec le nom et des parenthèses :

```
# ListStack.(push 5 empty);;  
- : int ListStack.t = <abstr>
```

En utilisant open :

```
# open ListStack;;  
# push 5 empty;;  
- : int ListStack.t = <abstr>
```

En général, on évite open pour diminuer les conflits de noms.

<abstr>

```
# ListStack.push 5 ListStack.empty;;  
- : int ListStack.t = <abstr>
```

<abstr> signifie qu'on n'a pas de contrôle sur la représentation interne

- on ne peut pas l'inspecter (sans passer par le module)
- on ne pas créer des valeurs (sans passer par le module)

C'est donc une valeur *abstraite*

Pipelining

L'opérateur `|>` peut être utile quand on souhaite appliquer des opérations chaînées, typiquement sur une structure de données :

```
# ListStack.(empty |> push 5 |> push 7 |> pop)
```

Pour cette raison, on va préférer prendre le type de donnée traité (ici, la pile) en **dernier argument**.

Structures de données fonctionnelles

Structures de données fonctionnelles

Notre structure de pile est fonctionnelle, ou *persistante* :

- pas d'effet de bord : on ne modifie jamais une pile
- on crée des nouvelles piles à chaque opération

```
# let x = ListStack.empty;;  
val x : 'a ListStack.t = <abstr>  
# let y = ListStack.push 5 x;;  
val y : int ListStack.t = <abstr>  
# ListStack.size x, ListStack.size y;;  
- : int * int = (0, 1)
```

Le module List

On a déjà utilisé les listes : `'a list`, et des fonctions du module `List`.

- `'a List.t` est `'a list`
- cf. `doc`
 - `#show List` dans UTop pour la signature

Type du module List

```
module List :  
  sig  
    type 'a t = 'a list = [] | (::) of 'a * 'a list  
    val length : 'a t -> int  
    val hd : 'a t -> 'a  
    val tl : 'a t -> 'a t  
    val nth : 'a t -> int -> 'a  
    val nth_opt : 'a t -> int -> 'a option  
    val rev : 'a t -> 'a t  
    val concat : 'a t t -> 'a t  
    val map : ('a -> 'b) -> 'a t -> 'b t  
    ...  
  end
```

Le module Option

Le type `option` permet de représenter la possibilité d'absence d'une valeur.

- `'a Option.t` est `'a option`
- cf. `doc`
 - `#show Option` dans `UTop` pour la signature

Type du module Option

module Option :

sig

type 'a t = 'a **option** = **None** | **Some** **of** 'a

val none : 'a t

val some : 'a -> 'a t

val value : 'a t -> default:'a -> 'a

val get : 'a t -> 'a

val bind : 'a t -> ('a -> 'b t) -> 'b t

val join : 'a t t -> 'a t

val map : ('a -> 'b) -> 'a t -> 'b t

val fold : none:'a -> some:('b -> 'a) -> 'b t -> 'a

val iter : ('a -> **unit**) -> 'a t -> **unit**

val is_none : 'a t -> **bool**

val is_some : 'a t -> **bool**

...

end

Réimplémentation de Option

Dans option.ml :

```
type 'a t = None | Some of 'a
```

```
let none : 'a t = None
```

```
let some (v : 'a) : 'a t = Some v
```

Réimplémentation de Option

(Récupère la valeur s'il y en a une, sinon prend une valeur par défaut *)*

```
let value ~(default : 'a) (opt : 'a t) : 'a =  
  match opt with  
  | Some v -> v  
  | None -> default
```

(Récupère la valeur s'il y en a une, sinon lève une exception *)*

```
let get (opt : 'a t) : 'a =  
  match opt with  
  | Some v -> v  
  | None -> raise Invalid_argument
```

Réimplémentation de Option

(Applique une fonction dans l'option *)*

```
let map (f : 'a -> 'b) (opt : 'a t) : 'b t =  
  match opt with  
  | Some v -> Some (f v)  
  | None -> None
```

(Applique une fonction qui peut "rater" *)*

```
let bind (opt : 'a t) (f : 'a -> 'b t) : 'b t =  
  match opt with  
  | Some v -> f v  
  | None -> None
```

Réimplémentation de Option

```
(* Applique une fonction qui retourne unit *)  
let iter (f : 'a -> unit) (opt : 'a t) : unit =  
  match opt with  
  | Some v -> f v  
  | None -> ()
```

Réimplémentation de Option

(Repli sur les options *)*

```
let fold ~(none : 'a) ~(some : 'b -> 'a) (opt : 'b t) -> 'a =  
  match opt with  
  | Some v -> some v  
  | None -> none
```

(Enlève un niveau d'imbrication *)*

```
let join (opt : 'a t t) : 'a t =  
  match opt with  
  | Some (Some v) -> Some v  
  | None -> None
```

Réimplémentation de Option

(Vérifie si une option est vide *)*

```
let is_none (opt : 'a t) : bool =  
  match opt with  
  | Some v -> false  
  | None -> true
```

(Vérifie si une option est remplie *)*

```
let is_some (opt : 'a t) : bool =  
  match opt with  
  | Some v -> true  
  | None -> false
```

STACK avec Option

Dans la spécification de STACK, on a utilisé des exceptions :

- peek sur la liste vide
- pop sur la liste vide

On peut utiliser des options à la place.

Cela requiert un changement de spécification.

STACK avec Option

Nouvelle spécification :

```
module type STACK = sig
  type 'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val push : 'a -> 'a t -> 'a t
  val peek : 'a t -> 'a option (* modifié *)
  val pop : 'a t -> 'a t option (* modifié *)
  val size : 'a t -> int
  val to_list : 'a t -> 'a list
end
```

ListStack avec Option

```
module ListStack : STACK = struct
  type 'a t = 'a list
  exception Empty
  let empty = []
  let is_empty = function [] -> true | _ -> false
  let push = List.cons
  let peek = function [] -> None | x :: _ -> Some x
  let pop = function [] -> None | _ :: s -> Some s
  let size = List.length
  let to_list = Fun.id
end
```

Pipelining

On ne sait plus faire de pipelining !

```
# ListStack.(empty |> push 1 |> pop |> peek);;
```

Error: This expression has **type** `int t option`
but an expression was expected **of type** `'a t`

map et bind

On peut récupérer une forme de pipelining avec deux opérateurs :

```
let ( >>| ) (opt : 'a option) (f : 'a -> 'b) : 'b option = Option.map f opt
let ( >>= ) : 'a option -> ('a -> 'b option) -> 'b option = Option.bind
```

- >>| est souvent appelé map (ou fmap), et est en lien avec le concept de *foncteur applicatif*
 - Option.map
- >>= est souvent appelé bind, et est en lien avec le concept de *monade*
 - Option.bind

Concepts cruciaux pour d'autres langages fonctionnels (Haskell), moins important pour ce cours.

Pipelining

```
# ListStack.(empty |> push 1 |> pop >>|  
              push 2 >>= pop >>| push 3 >>| to_list)  
- : int list option = Some [3]
```

Mais on a perdu en facilité d'utilisation... c'est un compromis.

Foncteurs applicatifs et monades

On peut utiliser des *extensions de langage* pour cacher les bind et map.

```
module OptionSyntax = struct
  let ( let+ ) opt f = Option.map f opt
  let ( let* ) = Option.bind
end
```

Foncteurs applicatifs et monades

```
# open OptionSyntax;;  
# let f (x : int option) : int option =  
    let+ x_value = x in  
    x_value + 1;;  
val f : int option -> int option = <fun>  
# f (Some 0);;  
- : int option = Some 1  
# f (Some 5);;  
- : int option = Some 6
```

C'est équivalent à :

```
let f (x : int option) : int option =  
    Option.map (fun x -> x + 1) x;;
```

Foncteurs applicatifs et monades

```
# let f (x : int option) (y : int option) : int option =  
    let* x_value = x in  
    let+ y_value = y in  
    x_value + y_value;;  
val f : int option -> int option -> int option = <fun>  
# f (Some 5) (Some 1);;  
- : int option = Some 6
```

Foncteurs applicatifs et monades

```
# let f (x : int option) (y : int option) : int option =  
  let* x_value = x in  
  let* y_value = y in  
  Some (x_value + y_value);;  
val f : int option -> int option -> int option = <fun>  
# f (Some 5) None;;  
- : int option = None  
# f (Some 5) (Some 1);;  
- : int option = Some 6
```

Foncteurs applicatifs et monades

```
# let+ x = Some 42 in x + 1;;  
- : int option = Some 43  
# let* x = Some 42 in Some (x + 1);;  
- : int option = Some 43
```

Foncteurs applicatifs et monades

```
# open ListStack;;  
# open OptionSyntax;;  
# let* l1 = empty |> push 1 |> pop in  
  let* l2 = push 2 l1 |> pop in  
  let l3 = push 3 l2 |> to_list in  
  Some l3 ;;  
- : int list option = Some [3]
```

Foncteurs applicatifs et monades

C'est un patron applicable à d'autres structures de données :

```
module ListSyntax = struct
  let ( let+ ) opt f = List.map f opt
  let bind (l : 'a list) (f : 'a -> 'b list) : 'b list =
    List.flatten (List.map f l)
  let ( let* ) = bind
end
```

Foncteurs applicatifs et monades

```
# open ListSyntax;;  
# let pairs (l1 : 'a list) (l2 : 'b list) : ('a * 'b) list =  
    let* x = l1 in  
    let* y = l2 in  
    [(x, y)];;  
val pairs : 'a list -> 'b list -> ('a * 'b) list = <fun>  
# pairs [1;2;3] [4;5;6];;  
- : (int * int) list =  
[(1, 4); (1, 5); (1, 6); (2, 4); (2, 5); (2, 6); (3, 4); (3, 5); (3, 6)]
```

Le module Result

Si on souhaite ajouter de l'information aux erreurs :

```
type ('a, 'e) result =  
| Ok of 'a  
| Error of 'e
```

- Ok x est similaire à Some x des options
- Error x est similaire à None des options, mais avec un message d'erreur

Voir la [documentation](#)

STACK avec Result

```
module type STACK = sig
  type 'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val push : 'a -> 'a t -> 'a t
  val peek : 'a t -> ('a, string) result
  val pop : 'a t -> ('a t, string) result
  val size : 'a t -> int
  val to_list : 'a t -> 'a list
end
```

ListStack avec Option

```
module ListStack : Stack = struct
  type 'a t = 'a list
  exception Empty
  let empty = []
  let is_empty = function [] -> true | _ -> false
  let push = List.cons
  let peek = function [] -> Error "peek on empty stack" | x :: _ -> Ok x
  let pop = function [] -> Error "pop on empty stack" | _ :: s -> Ok s
  let size = List.length
  let to_list = Fun.id
end
```

Exercice: définir map et bind sur result

Exercice

Définir map et bind sur le type result:

- donner leur signature
- compléter la définition

Exercice: définir map et bind sur result

```
let map (res : ('a, 'e) result) (f : 'a -> 'b) : ('b, 'e) result =  
  match res with  
  | Error e -> Error e  
  | Ok x -> Ok (f x)
```

```
let bind (res : ('a, 'e) result)  
  (f : 'a -> ('b, 'e) result) : ('b, 'e) result =  
  match res with  
  | Error e -> Error e  
  | Ok x -> f x
```

Gestion d'erreurs en OCaml

Rappel du chapitre 3 : il y a trois façons de gérer les erreurs en OCaml :

- utiliser le type `'a option`
- utiliser le type `'a result`
- utiliser les exceptions

On préférera le type `option`, ou les exceptions.

Le module Seq et l'évaluation paresseuse

Voir la [documentation](#) de Seq

Seq.t est un type de liste paresseuses :

- les éléments de la liste seront calculés quand on en aura besoin
- pour retarder leur évaluation, on les met dans une fonction

```
type 'a t = unit -> 'a node
```

```
type 'a node = Nil | Cons of 'a * 'a t
```

- Un 'a t est une fonction qui retourne un nœud de la liste
- Un nœud est composé d'un élément, et d'un 'a t pour calculer le reste de la liste

Listes paresseuses

```
# let rec numbers (n : int) : int Seq.t =  
    fun () -> Seq.Cons (n, (numbers (n + 1))) ;;  
val numbers : int -> int Seq.t = <fun>  
# numbers 0;;  
- : int Seq.t = <fun>
```

numbers est une liste infinie (paresseuse) des entiers à partir de 0.

```
# numbers 0 ();;  
- : int Seq.node = Seq.Cons (1, <fun>)
```

Listes paresseuses

```
# let even_numbers : int Seq.t =  
    Seq.filter (fun x -> x mod 2 = 0) (numbers 0) ;;  
# let square_of_even_numbers : int Seq.t =  
    Seq.map (fun x -> x*x) even_numbers  
# List.of_seq (Seq.take 10 square_of_even_numbers) ;;  
- : int list = [0; 4; 16; 36; 64; 100; 144; 196; 256; 324]
```

Listes paresseuses

```
let fac (n : int) : int Seq.t =  
  Seq.ints 1           (* tous les entiers à partir de 1 *)  
  |> Seq.scan ( * ) 1   (* calcule leur produit *)  
  |> Seq.drop n         (* enlève les n premiers éléments *)  
  |> Seq.uncons         (* transforme en paire *)  
  |> Option.get |> fst (* récupère le premier élément *)
```

Un module BinarySearchTree

On veut implémenter une structure d'arbre binaire de recherche.

```
module type BINARY_SEARCH_TREE = sig
  type 'a t
  val mem : 'a -> 'a t -> bool
  val insert : 'a -> 'a t -> 'a t
  val map : ('a -> 'b) -> 'a t -> 'b t
end
```

Un module BinarySearchTree

```
module BinarySearchTree : BINARY_SEARCH_TREE = struct
  type 'a t = Node of 'a t * 'a * 'a t | Leaf
  ...
end
```

Un module BinarySearchTree : mem

```
let rec mem (x : 'a) (tree : 'a t) : bool =  
  match tree with  
  | Leaf -> false  
  | Node (l, x', _) when x < x' -> mem x l  
  | Node (_, x', r) when x > x' -> mem x r  
  | _ -> true
```

BinarySearchTree : insert

```
let rec insert (x : 'a) (tree : 'a t) =  
  match tree with  
  | Leaf -> Node (Leaf, x, Leaf)  
  | Node (l, x', r) when x < x' -> Node (insert x l, x', r)  
  | Node (l, x', r) when x > x' -> Node (l, x', insert x r)  
  | _ -> tree
```

BinarySearchTree : map

```
let rec map (f : 'a -> 'b) (tree : 'a t) : 'b t =  
  match tree with  
  | Leaf -> Leaf  
  | Node (l, x, r) -> Node (map f l, f x, map f r)
```

Ensembles avec Set

On souhaite implémenter un module qui représente des ensembles.

Possibilités :

- on utilise des listes
 - pour ajouter un élément : il faut parcourir la liste en entier pour voir si l'élément est présent, $\mathcal{O}(n)$
 - pour extraire un élément : il faut parcourir la liste en entier, $\mathcal{O}(n)$
 - marche pour n'importe quel type de donnée qu'on peut comparer avec =
 - pas très efficace...
- on utilise une structure arborescente (e.g., arbre rouge-noir)
 - pour ajouter un élément : $\mathcal{O}(\log n)$
 - pour extraire un élément : $\mathcal{O}(\log n)$
 - il faut pouvoir comparer les éléments

Le module Set

OCaml implémente l'approche arborescente dans Set, mais il faut une relation d'ordre :

```
module Set : sig
  module type OrderedType = sig type t val compare : t -> t -> int end
  module type S =
    sig
      type elt
      type t
      val add : elt -> t -> t
      val remove : elt -> t -> t
      val union : t -> t -> t
      ...
    end
  module Make : functor (Ord : OrderedType) -> sig ... end
end
```

OrderedType

Le type de module OrderedType définit un type et une façon d'ordonner ses valeurs :

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

OrderedType pour int

```
module IntOrdered : OrderedType = struct
  type t = int
  let compare x y =
    if x < y then
      -1
    else if x > y then
      1
    else
      0
end
```

OrderedType pour int

Ou, plus simple :

```
module IntOrdered : OrderedType = struct
  type t = int
  let compare x y = Stdlib.compare x y
end
```

Car compare est déjà défini pour de nombreux types

OrderedType pour int

Ou encore :

```
module IntOrdered : OrderedType = Int
```

car Int contient les définitions qu'on souhaite

OrderedType pour `int * int`

```
module IntPair = struct
  type t = int * int
  let compare (x0,y0) (x1,y1) =
    match Int.compare x0 x1 with
    | 0 -> Int.compare y0 y1
    | c -> c
end
```

Set.Make

Pour créer un Set, il faut donc un OrderedType :

```
module IntSet = Set.Make(Int)
module PairsSet = Set.Make(IntPair)
```

On peut ensuite utiliser le module ainsi créé

Foncteur de module

```
module type S = sig
  type elt
  type t
  val add : elt -> t -> t
  ...
end
module Make(Ord: OrderedType) : S = struct
  type elt = Ord.t
  type t = ...
  let add = ...
  ...
end
```

Foncteur de module

`Set.Make` est un module d'ordre supérieur :

- il prend un module en argument
- il retourne un module

En OCaml, un module d'ordre supérieur est appelé un *foncteur* (différent des foncteurs applicatifs)

Aspect avancés des modules

Le système de module de OCaml est très puissant et contient de nombreuses fonctionnalités avancées :

- contraintes de types
- inclusion de modules dans un autre
- modules d'ordre supérieur
- construction dynamique de modules

Mais on se limitera principalement à un module comme étant un espace de nom permettant de faire de l'abstraction.

Concepts clés

- **Module** : élément de programmation modulaire, regroupe type de données et fonctions
 - `module X = struct ... end` en OCaml
- **Type abstrait** : type dont on ne connaît pas l'implémentation, `<abstr>`
 - en OCaml, c'est fait en donnant un type plus restreint au module
- **Encapsulation** : cacher les détails internes d'implémentation
- **Structure fonctionnelle** : structure immuable
- **Évaluation paresseuse** : on définit les choses généralement, mais on ne calcule qu'au besoin

Exemple : 2048