ManO (MU4IN511) : COURS 2

Mise à Niveau et Ouverture (STL)

Emmanuel Chailloux

Programmation impérative

- modèle plus proche des machines réelles
- tout est dans `X := X + 1`
  - exécution d'une instruction (action) qui modifie l'état mémoire
  - passage à une nouvelle instruction dans le nouvel état mémoire
- modèle des langages Fortan, Pascal, C, Ada, Rust, ...

Entrées/sorties

Canaux:

- types : *in_channel* et *out_channel*
- fonctions : `open_in` : *string*→ *in_channel* (`close_in`)
  `open_out` : *string*→*out_channel* (`close_out`)
- exception : *End_of_file*
- canaux prédéfinis : *stdin*, *stdout* et *stderr*
- fonctions de lecture et d'écriture sur les canaux
- organisation et accès séquentiels
- type *open_flag* pour les modes d'ouverture

Principales fonctions d'ES

`input`	:	*in_channel* → *bytes* → *int* → *int* → *int*
`input_line`	:	*in_channel* → *string*
`output`	:	*out_channel* → *bytes* → *int* → *int* → *unit*
`output_string`	:	*out_channel* → *string* → *unit*
`output_bytes`	:	*out_channel* → *bytes* → *unit*
`read_line`	:	*unit* → *string*
`read_int`	:	*unit* → *int*
`print_string`	:	*string* → *unit*
`print_bytes`	:	*bytes* → *unit*
`print_int`	:	*int* → *unit*
`print_newline`	:	*unit* → *unit*

string : chaînes immutables

bytes : chaînes mutables

unit : type ne possédant qu'une seule valeur `()`

Exemple : C+/C-

```
1  # let rec cpcm n =
2      let _ = print_string "taper un nombre : " in
3      let i = read_int () in
4        if i = n then print_string "BRAVO\n\n"
5        else let _ = (if i < n then print_string "C+\n"
6                      else print_string "C-\n")
7             in cpcm n;;
8  val cpcm : int -> unit = <fun>
9
10 # cpcm 64;;
11 taper un nombre : 88
12 C-
13 taper un nombre : 44
14 C+
```

Valeurs physiquement modifiables

- valeurs structurées dont une partie peut être physiquement (en mémoire) modifiée ;
- vecteurs, enregistrements ou variants à champs modifiables, bytes, références

$\Rightarrow$ nécessite de contrôler l'ordre du calcul ! ! !

Attention: l'ordre d'évaluation des arguments n'est pas spécifié.

Vecteurs (1)

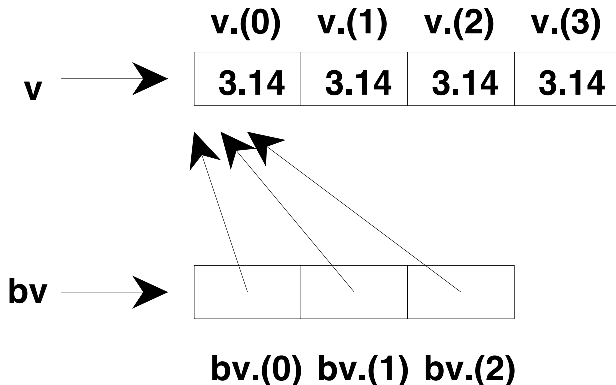
- regroupent un nombre connu d'éléments de même type
- création : `Array.create` : $int \rightarrow$ 'a $\rightarrow$ 'a array,
- longueur : `Array.length` : 'a array$\rightarrow$ int
- accès : $e_1.(e_2)$
- modification : $e_1.(e_2)$<-e_3

Vecteurs (2)

```
1  # let v = Array.create 4 3.14;;
2  val v : float array = [|3.14; 3.14; 3.14; 3.14|]
3
4  # v.(1);;
5  - : float = 3.14
6
7  # v.(8);;
8  Exception: Invalid_argument "Array.get".
9
10 # v.(0) <- 100.;;
11 - : unit = ()
12
13 # v;;
14 - : float array = [|100.; 3.14; 3.14; 3.14|]
```

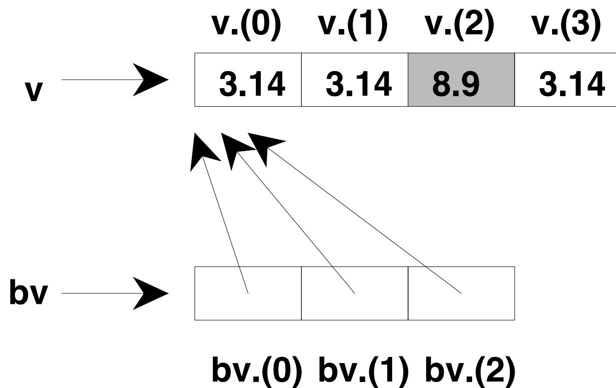
Représentation mémoire (1)

```
1  # let bv = Array.create 3 v;;
```



Représentation mémoire (2)

```
1  # v.(2) <- 8.9;;
```




```
1  # if bv.(1).(2) =  8.9 then "A" else "B";;
```

Fonctions sur les vecteurs

- création matrice
  - `Array.make_matrix` : *int → int → 'a → 'a array array*
- itérateurs
  - `iter` : *('a → unit) → 'a array → unit*
  - `map` : *('a → 'b) → 'a array → 'b array*
  - `iteri` : *(int → 'a → unit) → 'a array → unit*
  - `mapi`, `fold_left`, `fold_right`, `...`

Enregistrements à champs mutables

- indication à la déclaration de type d'un champs est "mutable"
- accès identique $e_1.f_i$, modification $e_1.f_i$<-e_2

type t $=${*f1 : t1* ; **mutable** *f2 :t2* ; ... ; *fn :tn*}

```
1  # type point = {mutable x : float; mutable y : float};;
2  type point = { mutable x: float; mutable y: float }
3  # let p = {x=1.; y=1.};;
4  val p : point = {x=1; y=1}
5  # p.x <- p.x +. 1.0;;
6  - : unit = ()
7  # p;;
8  - : point = {x=2; y=1}
```

Bytes et Chaînes de caractères

- les chaînes ne sont plus des valeurs modifiables depuis la version 4.06 (fonction `input`), on utilise alors les *bytes* avec `set` et `get`.
- accès avec `Bytes.get` : `bytes -> int -> char`
- modification avec `Bytes.set` = `bytes -> int -> char -> unit`

```
1  # let s  = Bytes.of_string "bonjour";;
2  val s : bytes = Bytes.of_string "bonjour"
3  # Bytes.get s 3;;
4  - : char = 'j'
5  # Bytes.set s 3 '-' ;;
6  - : unit = ()
7  # s;;
8  - : bytes = Bytes.of_string "bon-our"
```

Depuis la vers 4.06 d'OCaml les valeurs de type string sont immutables.

Références

- sous-cas historique utilisant maintenant des records mutables
  - **type** 'a ref = {**mutable** contents:'a}
  - $!e_1 \quad \equiv \quad e_1$.contents
  - e_1:=$e_2 \quad \equiv \quad e_1$.contents<-$e_2$

```
1  # let incr x = x := !x + 1;;
2  val incr : int ref -> unit = <fun>
3  # let z = ref 3;;
4  val z : int ref = {contents=3}
5  # incr z;;
6  - : unit = ()
7  # z;;
8  - : int ref = {contents=4}
9  # (ref 3) := 2;;
```

Structures de contrôle

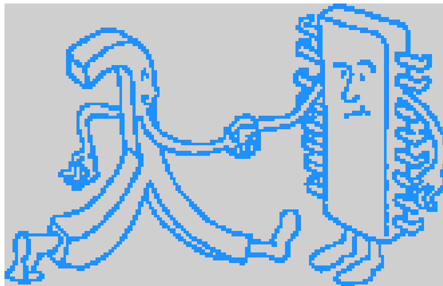
- séquentielle : $e1$; e_2 ; ... ; e_n
  regroupée : (...) ou **begin** ... **end**
  le type de la séquence est le type de e_n
  (dernière expression de la séquence)
- conditionnelle : **if** c_1 **then** e_2 (e_2 de type *unit*)
- itératives :
  - **while** c **do** e **done**
  - **for** v=e_1 [**down**]**to** e_2 **do** e_3 **done**

La conditionnelle et les boucles sont des expressions de type *unit*

Exemple : somme de 2 vecteurs

```
1   #let somme a b =
2     let al = Array.length a  and bl = Array.length b in
3     if al <> bl then failwith "somme"
4     else  if al = 0 then a
5       else
6         let c = Array.create al a.(0) in
7           for i=0 to al-1 do
8             c.(i) <- a.(i) + b.(i)
9           done;
10          c;;
11  val somme : int array -> int array -> int array = <fun>
12  # somme [|1; 2; 3|] [| 9; 10; 11|];;
13  - : int array = [|10; 12; 14|]
```

Style fonctionnel-impératif



Style fonctionnel ou impératif

- utiliser le bon style selon les structures de données et leurs manipulations (par copie ou en place)
  - impératif sur les matrices (en place)
  - fonctionnel sur les arbres (par copie)
- mélanger les deux styles
  - exemple : calcul de distances
  - valeurs fonctionnelles modifiables
  - implantation de l'évaluation retardée

Fonction : map

- style fonctionnel

```
1  # let rec fmap f l = match l with
2    [] -> []
3  | h::t -> let r = f h in r::(fmap f t);;
4  val fmap : ('a -> 'b) -> 'a list -> 'b list = <fun>
5
6  #  fmap (function x -> x + 1)  [3; 1; 0];;
7  - : int list = [4; 2; 1]
```

- style impératif

```
1  # let imap f l  =
2    let nl = ref l
3    and nr = ref [] in
4      while (!nl <> []) do
5        nr := ( f (List.hd !nl)) :: (!nr);
6        nl := List.tl !nl
7      done;
8      List.rev !nr;;
9  val imap : ('a -> 'b) -> 'a list -> 'b list = <fun>
10
11 # imap (function x -> x + 1)  [3; 1; 0];;
12 - : int list = [4; 2; 1]
```

Transposée de matrice (1)

- style impératif

```
1  # let itrans m = let l = Array.length m in
2      for i=0 to l-1 do
3        for j=i to l-1 do
4          let v = m.(i).(j) in
5            m.(i).(j) <- m.(j).(i);
6            m.(j).(i) <- v
7        done done;;
8  val itrans : 'a array array -> unit = <fun>
9
10 # let v = [| [| 1; 2; 4|]; [|3; 3; 3|]; [|9; 8; 7|] |];;
11 val v : int array array = [|[|1; 2; 4|]; [|3; 3; 3|]; [|9; 8; 7|]|]
12 # itrans v;;
13 - : unit = ()
14 # v;;
15 - : int array array = [|[|1; 3; 9|]; [|2; 3; 8|]; [|4; 3; 7|]|]
```

Transposée de matrice (2)

- style fonctionnel

```
1  # let rec  ftransl l = match l with
2    []::_ -> []
3  | _ ->   (List.map List.hd l) ::
4           ftransl (List.map List.tl l);;
5           val ftransl : 'a list list -> 'a list list = <fun>
6
7  # ftransl [ [1; 2; 4]; [3; 3; 3]; [9; 8; 7] ];;
8  - : int list list = [[1; 3; 9]; [2; 3; 8]; [4; 3; 7]]
```

Exemple : calcul de distance (1)

distance Manhattan : +1 par bloc horizontal ou vertical,
ici avec une structure de tore (chambre à air)

```
# let vide = -1 ;;
val vide : int = -1
# let mur = min_int;;
val mur : int = -4611686018427387904
# let taille = 9 ;;
val taille : int = 9
# let monde = Array.make_matrix taille taille vide ;;
val monde : int array array =

# let norme x = (x + taille) mod taille ;;
val norme : int -> int = <fun>
# let rec dist (px,py) d m =
  let npx = norme px and npy = norme py in
  let v = m.(npx).(npy) in
    if v = vide || v > d then (
      m.(npx).(npy) <- d ;
      dist (px+1,py) (d+1) m ;
      dist (px-1,py) (d+1) m ;
      dist (px,py+1) (d+1) m ;
      dist (px,py-1) (d+1) m ) ;;
val dist : int * int -> int -> int array array -> unit = <fun>
```

Exemple : calcul de distance (2)

```
# monde.(2).(4) <- mur ; monde.(3).(4) <- mur ; monde.(2).(5) <- mur ;;
- : unit = ()
# dist (2,3) 0 monde ;;
- : unit = ()
# let affiche m =
  Array.iter (fun v ->
    Array.iter (fun w ->
      (if w = mur then print_string "." else print_int w);
      print_string " ") v; print_newline()) m ;;
val affiche : int array array -> unit = <fun>
# affiche monde ;;

5 4 3 2 3 4 5 6 6
4 3 2 1 2 3 4 5 5
3 2 1 0 . . 5 5 4
4 3 2 1 . 5 6 6 5
5 4 3 2 3 4 5 6 6
6 5 4 3 4 5 6 7 7
7 6 5 4 5 6 7 8 8
7 6 5 4 5 6 7 8 8
6 5 4 3 4 5 6 7 7
- : unit = ()
```

Exemple : calcul de distance (3)

- en récursif terminal, en utilisant un file d'attente

```
1  # let  dist (px,py) d m =
2     let q = Queue.create () in
3     let rec aux (px,py,d)  =
4       let npx = norme px and npy = norme py in
5       let v = m.(npx).(npy) in
6       if v = vide || v > d then (
7        m.(npx).(npy) <- d ;
8        Queue.add (px+1,py,d+1) q ;
9        Queue.add (px-1,py,d+1) q ;
10       Queue.add (px,py+1,d+1) q ;
11       Queue.add (px,py-1,d+1) q ) ;
12      if (not(Queue.is_empty q)) then aux (Queue.take q)
13    in
14      aux (px,py,d) ;;
15 val dist : int * int -> int -> int array array -> unit = <fun>
```

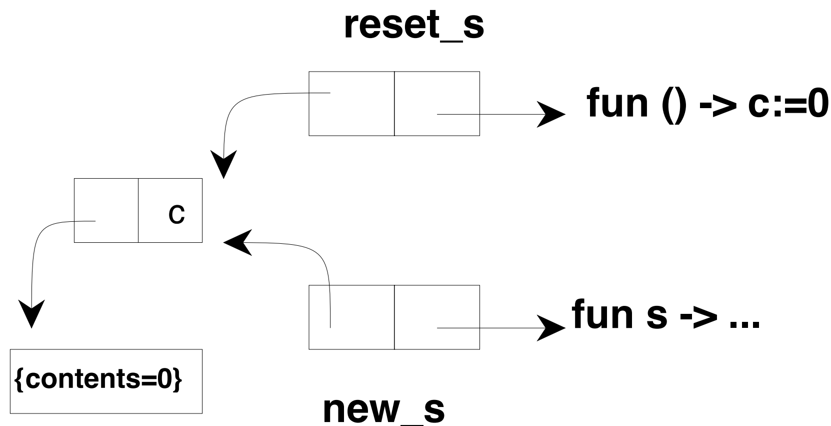
Représentation des fermetures

- couple : code - environnement
  - code : adresse mémoire du **function** compilé
  - environnement : contient les valeurs des variables libres non globales du corps du **function**
- connaissance à la compilation de la position d'une variable dans l'environnement
- permet l'extension de portée d'une déclaration locale

Générateur de symboles

```
1  #   let reset_s,new_s = let c = ref 0 in
2     ( function () -> c := 0),
3     ( function s ->  c:=!c+1; s^(string_of_int !c));;
4  val reset_s : unit -> unit = <fun>
5  val new_s : string -> string = <fun>
6
7  # new_s "VAR";;
8  - : string = "VAR1"
9  # new_s "VAR";;
10 - : string = "VAR2"
11
12 # reset_s();;
13 - : unit = ()
14 # new_s "WAR";;
15 - : string = "WAR1"
```

Représentation mémoire



Résumé des expressions en OCaml

```
1   expr ::= ...
2          | [| expr; expr; ...; expr |]
3          | expr.(expr) | expr.(expr) <- expr
4          | expr.champs  | expr.champs <- expr
5          | ref expr | !expr | expr := expr
6          | expr.[expr] | expr.[expr] <- expr
7          | expr ; ... ; expr
8          | if expr then expr
9          | for var = expr to expr do expr done
10         | for var = expr downto expr do expr done
11         | while expr do expr done
```

Programmation modulaire

- découpage en *unités logiques* plus petites ;

But: réalisation d'un module séparément des autres modules

Mise en œuvre: un module possède une *interface*, la vérification des interface est effectuée à l'assemblage des différents modules.

Intérêts:

- découpage logique ;
- abstraction des données (spécification et réalisation) ;
- indépendance de l'implantation ;
- réutilisation.

Compilation séparée

- découpage en *unités de compilation*, compilables séparément

programmation modulaire $\neq$ compilation séparée

les 2 approches sont nécessaires:

- Pour cela la spécification d'un module doit être vérifiable par un compilateur :
  - on se limite à la vérification de types
  - l'interface sera spécification de modules
  - et contiendra l'information de typage et de compilation pour les autres modules

Langage de modules d'OCaml

2 parties:

- *structure* : pour la partie réalisation/implantation
- *signature* : pour la partie spécification/interface

Le langage de modules est indépendant du langage de base.

Parallèle entre: le langage de base (*valeur : type*)
et le langage de module (*structure : signature*) ! ! !

Modules simples

Implantation: d'un module est une suite de définitions

- de valeurs y compris fonctionnelles
- de types
- d'exceptions
- de sous-modules

Spécification: d'un module est une suite de déclarations et de spécifications de types.

Notation: une signature sera écrite en MAJUSCULE et une structure en Minuscule dont l'initiale est en majuscule.

Implantation d'un module Queue

```ocaml
module Queue =

struct

 type 'a t = 'a list ref

 let create() = ref []

 let enq x q = q:= !q@[x]

 let deq q =
  match !q with
     [] -> failwith"Empty"
   | h::r -> q:=r; h

 let length q = List.length !q

end ;;
```

Synthèse d'une signature

L'exemple précédent donne la signature suivante :

```
1  module Queue :
2  sig
3    type 'a t = 'a list ref
4    val create : unit -> 'a list ref
5    val enq : 'a -> 'a list ref -> unit
6    val deq : 'a list ref -> 'a
7    val length : 'a list ref -> int
8  end
```

Modules : déclarations encapsulées

modules simples (structures)	leurs types (signatures)
ensemble de définitions	ensemble de spécifications de types

```
1  module Example =                (* signature inferee *)
2  struct                           sig
3    type t = int                     type t = int
4    module M =                       module M :
5     struct                           sig
6      let succ x = x+1                 val succ : int -> int
7     end                              end
8    let two = M.succ(1)              val two : int
9  end                              end
```

Accès aux éléments d'un module (1)

L'accès à un élément d'un module se fait par la notation "point".

```
1  # Queue.enq;;
2  - : 'a -> 'a list ref -> unit = <fun>
```

Y compris pour les champs d'enregistrements :

```
1  #  module Toto =   struct type t = {x:int; y:int}  end;;
2  module Toto : sig type t = { x: int; y: int } end
3  # let u = {Toto.x=3; Toto.y=18};;
4  val u : Toto.t = {Toto.x=3; Toto.y=18}
```

Ce qui peut être simplifié par l'**ouverture** du module :

```
1  # open Queue;
2  # let q = Queue.create() in ( enq "Bob" q; q);;
3  - : string list ref = {contents = ["Bob"]}
```

Accès aux éléments d'un module (2)

Exemple:

```
# Example.two;;
- : int = 2

# Example.M.succ;;
- : int -> int = <fun>

# Example.M.succ (Example.two);;
- : int = 3
```

Ouverture locale: : Module.(...)

```
# Example.(M.succ two + two) ;;
- : int = 5
# M.succ ;;
Error: Unbound module M
```

Déclaration d'une signature

```
module type QUEUE =
  sig
    type 'a t = 'a list ref
    val create : unit -> 'a list ref
    val enq : 'a -> 'a list ref -> unit
    val deq : 'a list ref -> 'a
    val length : 'a list ref -> int
  end
```

Quand une signature est associée à une structure il y a vérification de la cohérence :

- les déclarations de la signature existent dans la structure
- et satisfont les spécifications de la signature.

```
module Queue : QUEUE = struct ...   end;;
```

signatures

La signature ABS n'exporte pas :

- la représentation du type t,
- le module interne M.

```
1  module Example =                module type ABS =
2  struct                          sig
3    type t = int                    type t
4    module M =
5       struct ... end
6    let two = M.succ(1)              val two : t
7  end                             end
```

Rectriction par une signature

Le nouveau module `Abs` est une *vue restreinte* de `Example` : il montre les composants de l'interface `ABS`.

```
1  # module Abs = (Example : ABS);;
2
3  # Abs.two;;     (* t devient abstrait  *)
4  - : Abs.t = <abstr>
5
6  # Abs.M.succ;;  (* M est cache'  *)
7  Unbound value Abs.M.succ
```

Syntaxe du langage de modules (1)

- valeur : `module Nom [ : SIGNATURE ] =`

```
1        struct
2            let ...
3            type ...
4            exception ...
5            module ...
6        end
```

- type : `module type NOM =`

```
1        sig
2            val ...
3            type ...
4            exception ...
5            module ...
6        end
```

Syntaxe du langage de modules (2)

- abstraction (valeur fonctionnelle)

```
module Nom =
functor ( Module : SIGNATURE) ->
  struct ...
  end
```

- application

```
  module Nom = Module(Structure)
```

Syntaxe du langage de modules (3)

- déclaration de modules récursifs

```
module rec Nom [ : SIGNATURE ] =

  struct ... end

and  Nom [ : SIGNATURE ] =

  struct ... end

...
```

Attention, limitation sur les dépendances croisées de calcul.

Communication entre modules

Utilisation: de déclarations d'autres modules celle-ci peut être effectuée de 2 manières.

- **communication implicite:** en utilisant la notation “point” et en tenant compte de l'environnement global
- **communication explicite:** en utilisant des foncteurs (modules paramétrés par d'autres modules).

Communication implicite

```ocaml
module Element =  struct  type t = int  end;;
```

```ocaml
module QueueV2 =
  struct
    type element = Element.t

    type queue = element list ref

    exception Empty

    let create() = ((ref []) : queue)

    let enq x (q:queue) = q:= !q@[x]

    let deq (q:queue)  =
      match !q with
        [] -> raise Empty
      | h::r  -> q:=r; h
  end;;
```

Signature

```
1  # module type QUEUEV2 =
2    sig
3      type element = Element.t
4      and queue = element list ref
5      exception Empty
6      val create : unit -> queue
7      val enq : element -> queue -> unit
8      val deq : queue -> element
9    end;;
10
11 # module QueueV3 = (QueueV2 : QUEUEV2);;
12 module QueueV3 : QUEUEV2
13
14 # let q = QueueV3.create() in (QueueV3.enq 18 q; q);;
15 - : QueueV3.queue = {contents = [18]}
```

Paramétrisation et liens

modules paramétrés (foncteurs)

fonctions sur les modules

liens des modules

foncteurs appliqués

```
1  module FunEx =                                              FunEx(Example)
2    functor (X : ABS) ->
3      struct val p = X.two .. end
```

Communication explicite

```
module type ELEMENT =   sig  type t  end;;
```

```
module QueueFunc = functor (Element : ELEMENT) ->
    struct
    type element = Element.t

    type queue = element list ref

    exception Empty

    let create() = ((ref []) : queue)

    let enq x (q:queue) = q:= !q@[x]

    let deq (q:queue)  =
      match !q with
        [] -> raise Empty
      | h::r  -> q:=r; h
  end;;
```

Application d'un foncteur

```
# module QueueV4 = QueueFunc(Element);;
module QueueV4 : sig ... end

#  let q = QueueV4.create() in ( QueueV4.enq 44 q; q);;
- : QueueV4.queue = {contents = [44]}
```

```
# module NouvelElement =
  struct   type t = float end;;
module NouvelElement : sig type t = float end

# module QueueV5 = QueueFunc(NouvelElement);;
module QueueV5 :  sig ... end

#  let q = QueueV5.create() in ( QueueV5.enq 12.2 q; q);;
- : QueueV5.queue = {contents = [12.2]}
```

Exemple : module paramétré Set.Make

```
module type OrderedType =
  sig
    type t
    val compare: t -> t -> int
    end
module Make(Ord: OrderedType) =
  struct
    type elt = Ord.t
    type t = Empty | Node of t * elt * t * int
    (* ... *)
    let rec min_elt = function
        Empty -> raise Not_found
      | Node(Empty, v, _, _) -> v
      | Node(l, _, _, _) -> min_elt l
  end

  module IntPairs = struct
    type t = int * int
    let compare (x0,y0) (x1,y1) = match Pervasives.compare x0 x1 with
       0 -> Pervasives.compare y0 y1 | c -> c
  end
  module PairSet = Set.Make(IntPairs)
  let m = PairSet.(empty |> add (5,7) |> add (2,3) |> add (11,13))
  PairSet.min_elt m
```

Abstraction de types

Déclarations de types:

- concrètes (définition de type visible)
- abstraites (représentation du type masquée)

Intérêts de l'abstraction de types:

- indépendance de l'implantation
- limitation du polymorphisme

Exemple d'abstraction de types

2 Définitions pour les chaînes:

```
module type CHAINEGEN =
  sig  type t  val create : string -> t  end;;

module Chaine : CHAINEGEN = struct
  type t = string
  let  create (s:string) = ((String.copy s):t) end;;

module Maj  : CHAINEGEN = struct
  type t = string
  let  create (s:string) = ((String.uppercase s):t) end;;
```

```
#  let c1 = Chaine.create "salut";;
val c1 : Chaine.t = <abstr>

#  let m1 = Maj.create("salut");;
val m1 : Maj.t = <abstr>
```

Suite de l'exemple

```
1  # c1 = m1;;
2  This expression has type Maj.t
3   but is here used with type Chaine.t
```

```
1  # module QueueChaine  = QueueFunc(Chaine);;
2  # module QueueMaj = QueueFunc(Maj);;
```

```
1  # let q = QueueChaine.create() in
2      ( QueueChaine.enq (Chaine.create "Alicia") q; q);;
3  - : QueueChaine.queue = {contents = [<abstr>]}
4
5  #  QueueChaine.enq (Maj.create "Bob") q;;
6  This expression has type Maj.t but is here used with type
7    QueueChaine.element = Chaine.t
```

Compilation séparée (1)

Unité de compilation: 2 *fichiers*

- 1 fichier d'interface (`.mli`) + 1 fichier d'implantation (`.ml`)

Sans précision:

```
module Nom = (
   struct
     contenu du fichier nom.ml
   end :
   sig
     contenu du fichier nom.mli
   end)
```

Correspondance: nom de module et nom de fichier

- module `Nom` correspond aux fichiers : `nom.ml` et `nom.mli`
- environnement de typage : répertoires d'accès aux fichiers

Compilation séparée (2)

fichier interface: : `queue.mli`

```
1  type 'a t
2  exception Empty
3  val create : unit -> 'a t
4  val add : 'a -> 'a t -> unit
5  val push : 'a -> 'a t -> unit
6  val take : 'a t -> 'a
7  val pop : 'a t -> 'a
8  val peek : 'a t -> 'a
9  val top : 'a t -> 'a
10 val clear : 'a t -> unit
11 val copy : 'a t -> 'a t
12 val is_empty : 'a t -> bool
13 val length : 'a t -> int
14 val iter : ('a -> unit) -> 'a t -> unit
15 val fold : ('b -> 'a -> 'b) -> 'b -> 'a t -> 'b
16 val transfer : 'a t -> 'a t -> unit
```

Compilation séparée (3)

fichier implantation: : `queue.ml`

```
1   exception Empty
2
3   type 'a cell = { content: 'a; mutable next: 'a cell }
4   type 'a t = { mutable length: int; mutable tail: 'a cell }
5   let create () = { length = 0; tail = Obj.magic None }
6   let clear q = q.length <- 0; q.tail <- Obj.magic None
7   let add x q =
8     q.length <- q.length + 1;
9     if q.length = 1 then
10      let rec cell = { content = x; next = cell } in
11      q.tail <- cell
12    else
13      let tail = q.tail in
14      let head = tail.next in
15      let cell = { content = x; next = head } in
16      tail.next <- cell; q.tail <- cell
17
18  let push = add
```

Compilation séparée (4)

Compilation:

```
$ ocamlc -c queue.mli
$ ocamlc -c queue.ml
```

Fichiers objet:

```
$ ls queue.cm?
queue.cmi       queue.cmo
```

Compilation séparée (5)

utilisation:

```
let q = Queue.create();;
let r = Queue.create();;

let main() =
  Queue.add 3 q ; Queue.add 4 q ;
  Queue.add "Ping" r; Queue.add "Pong" r;
  print_int (Queue.take q); print_int (Queue.take q); print_newline();
  print_string (Queue.take r); print_string (Queue.take r); print_newline();;

main();;
```

compilation:

```
$ ocamlc queue.cmo main.ml -o main.exe
```

Exécution:

```
$ ./main.exe
34
PingPong
```

Ouverture d'un module

- global

  Syntaxe: **open** mod-name ; ;

  Racourci: de la notation "point"

  Exemple:

```
1 # open QueueV9;;
2 # let q = create();;
3 val q : QueueV9.queue = <abstr>
```

- local

  Syntaxe: **let open** mod-name **in** expr

Héritage d'un module par inclusion

Syntaxe: **include** mod-expr ; ;

réexportation dans la structure courante des définitions de `mod-expr`

Exemple:

```
1  module QueueV2B = struct
2    include QueueV2
3    let length (q : queue) = List.length !q
4  end;;
5
6  module QueueV2B :
7    sig
8      type element = Element.t
9      type queue = element list ref
10     exception Empty
11     val create : unit -> queue
12     val enq : element -> queue -> unit
13     val deq : queue -> element
14     val length : queue -> int
15   end
```

Différence entre `open` et `include`

- `open` crée des raccourcis des chemins des définitions d'une structure sans rien définir localement ;
- `include` ajoute les définitions du module inclus dans les définitions du module courant (héritage)

```
1  module QueueV2C = struct
2    open QueueV2
3    let create = create
4    let enq = enq
5    let deq = deq
6    let length (q : queue) = List.length !q
7  end;;
8
9  module QueueV2C :
10   sig
11     val create : unit -> QueueV2.queue
12     val enq : QueueV2.element -> QueueV2.queue -> unit
13     val deq : QueueV2.queue -> QueueV2.element
14     val length : QueueV2.queue -> int
15   end
```

Sous-modules

Définitions: de modules dans un module

Intérêts: organisation hiérarchique, visibilité des champs des modules extérieurs, nécessaire avec la compilation séparée

Exemple:

```
module M1 =
  struct  type t1
    let f = ...
    module type SMT1 = sig type t2 = (t1,t1) ... end
    module SM2 : SMT1 = struct let g x = f(f x) ... end
  end;;
```

Accès:

```
M1.SM2.g;;
```

Modules locaux

Syntaxe: **let module** mod-name = mod-expr in expr

Intérêt: création dynamique (à l'exécution) de modules

Exemple: appliquer un foncteur sur une *structure* dont l'un des champs est un paramètre d'une fonction.

```
1  # let g  (l : string list) =
2    let module Toto =
3      Set.Make(struct type t = string
4        let  compare a b = if a.[0] < b.[0] then -1
5          else if a.[0] > b.[0] then 1 else 0  end)
6    in Toto.min_elt
7      (List.fold_right Toto.add l Toto.empty);;
8  val g : string list -> string = <fun>
```

Foncteurs de foncteurs

Plusieurs paramètres: à un module paramétré

Intérêts: paramétrage d'un module application par plusieurs modules, création de squelettes, y compris avec abstraction de types et contraintes de partage.

Exemple: jeu à 2 joueurs

```
module Jeu =
  functor (Rep : REPRESENTATION) ->
  functor (Aff : AFFICHAGE) ->
  functor Alphabeta : ALPHABETA ->   struct ... end;;
module Main =  Jeu (Stone_rep) (Stone_graph (Stone_rep))
                    (Alphabeta (Stone_rep)) ;;
Main.main() ;;
```

Modules récursifs

déclaration **and** entre modules :

```
module rec A : sig ... end = struct ... end
and B : sig ... end = struct ... end
```

pour résoudre le cycle, cette déclaration nécessite au moins un module où l'on peut calculer toutes les valeurs. Pour simplifier il faut un module dit "safe" où toutes les valeurs ont un type fonctionnel (et qui peuvent alors être calculées). L'évaluation de ces modules commencent alors par le(s) module(s) "safe" .

voir exemple du manuel de référence : `http://caml.inria.fr/pub/docs/manual-ocaml/extn.html#sec220`

Pour aller plus loin (1)

- Sur le langage OCaml
  - extension objet : sous-typage structurel, polymorphisme de rangées :
  - typage : types sommes extensibles, variants polymorphes, GADT
  - modules : récursifs, de 1ère classe, ...
  - interopérabilité : C, JS, Java
  - mélange de styles : fonctionnel/impératif, et foncteur/classe,
- et son implantation : machine virtuelle, gestionnaire mémoire

$\Rightarrow$: cours MPIL : http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2018/ue/3I008-2019fev/

+ cours PCOMP :

http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2020/ue/LU3IN032-2021fev/

+ cours Compil : http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2018/ue/3I018-2019fev/

+ cours CA : https://www-apr.lip6.fr/ chaillou/Public/enseignement/2018-2019/ca/

Pour aller plus loin (2)

- Sur cette famille de langages
  - Reason : couche syntaxique d'OCaml (même compilateur), syntaxe pour le programme JS, interopérabilité avec JS
  - ReScript, évolution de BuckleScript, interopérabilité avec JS
  - F# : noyau fonctionnel, impératif d'OCaml, ajout d'une couche objet à la C#/.NET,
  - Swift : langages d'instructions, fonctionnel, typé statiquement, type somme et filtrage de motifs
  - SML (Standard ML) : autre langage issu de ML (Mlton, smlnj)
  - fonctionnel pur typé statiquement :
    - Haskell (fonctionnel pur) : évaluation retardée, introduit un style monadique

$\Rightarrow$: voir pointeurs Bibliographie

Pour aller plus loin (3)

- Influences entre les langages :
  - fusion des modèles fonctionnel et objet
    - Scala : implicites (pour la surcharge) et GADT
  - typage :
    - Rust : système de types garantissant la libération mémoire
    - TypeScript, ReScript et Flow : JS typé
    - Hack : Php typé
  - extensions langages main stream :
    - Java/C# : λ-expression, polymorphisme paramétrique (généricité), streams

$\Rightarrow$: traduire vos programmes en OCaml et réciproquement