Fonctionnalité et Modularité

Cours 6

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-fm

Plan

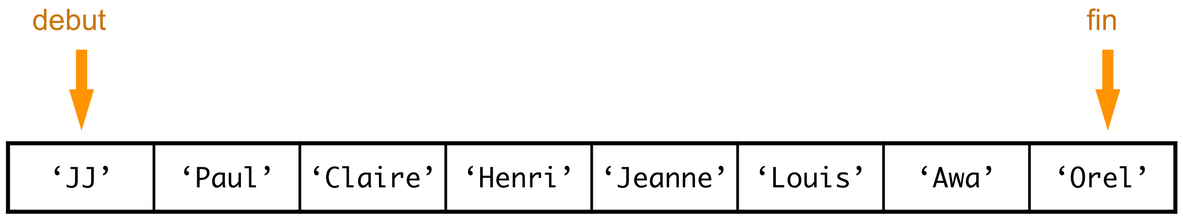
- modules
- signatures
- foncteurs
- exemples de modules
- typage fort
- compilation séparée

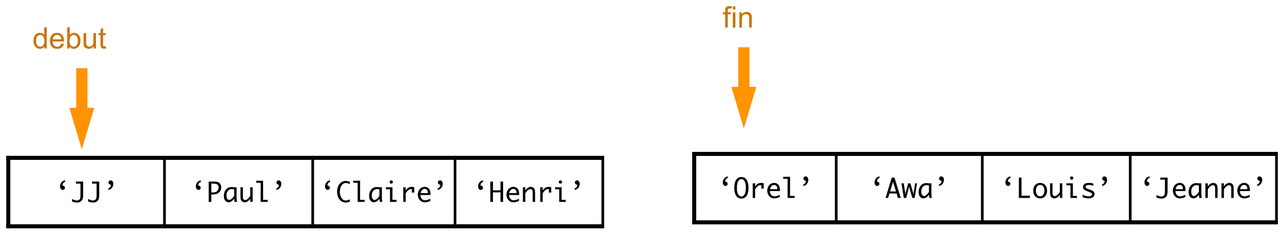
télécharger Ocaml en `http://www.ocaml.org`

Module

- déclaration d'un module pour les files d'attente

```
module Fifo1 =
    struct
      type 'a queue = { debut: 'a list; fin: 'a list }
      let make debut fin =
        match debut with
        | [ ] -> { debut = List.rev fin; fin = [ ] }
        | _  -> { debut; fin }
      let empty = { debut = [ ]; fin = [ ] }
      let is_empty = function { debut = [ ]; _ } -> true | _ -> false
      let add x q = make q.debut (x :: q.fin)
      exception Empty
      let fst = function
        | { debut = [ ]; _ } -> raise Empty
        | { debut = x :: _; _ } -> x
      let pop = function
        | { debut = [ ]; _ } -> raise Empty
        | { debut = _ :: d; fin = f } -> make d f
    end;;
```





- accès avec la notation qualifiée

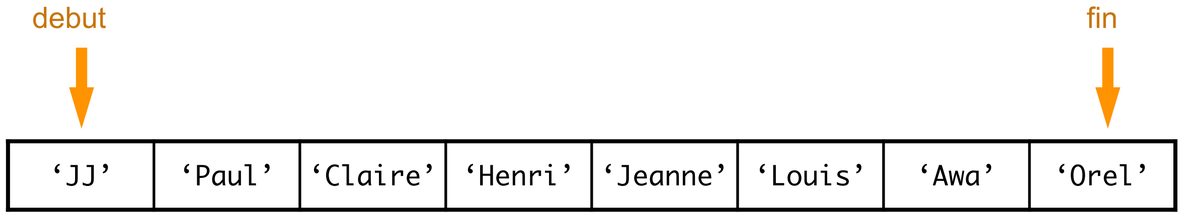
```
let q = Fifo1.empty ;;
let q1 = Fifo1.add "JJ" q ;;
let q2 = Fifo1.add "Paul" q1 ;;
let q3 = Fifo1.add "Claire" q2 ;;
(* - : string = "JJ" *)
let q4 = Fifo1.pop q3 ;;
(* - : string = "Paul" *)
```

les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

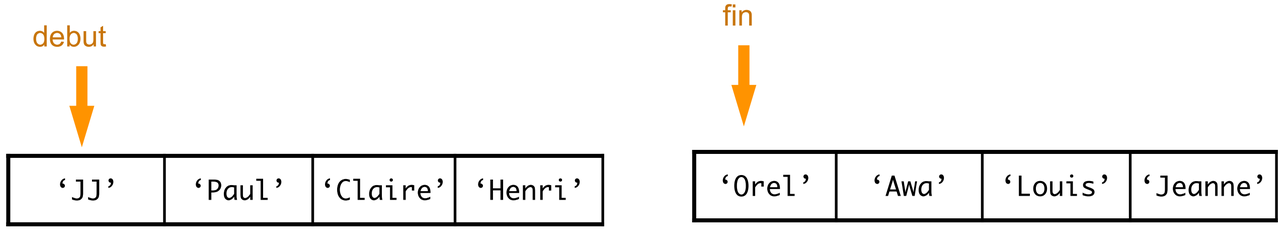
Module

- on peut ouvrir l'espace des noms du module

```
open Fifo1 ;;
let q = empty ;;
let q1 = add "JJ" q ;;
let q2 = add "Paul" q1 ;;
let q3 = add "Claire" q2 ;;
let q4 = pop q3 ;;
```



- dangereux car possibles collisions avec noms d'autres modules



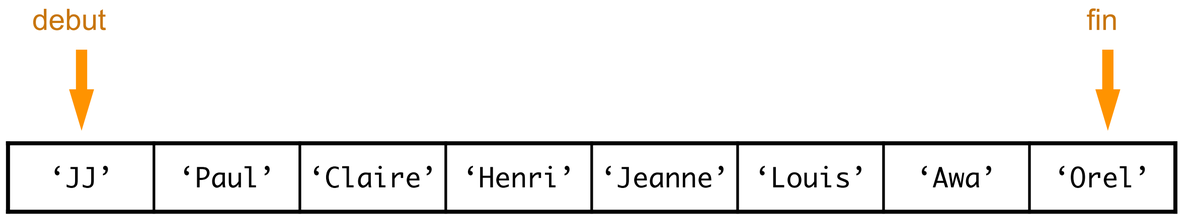
les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

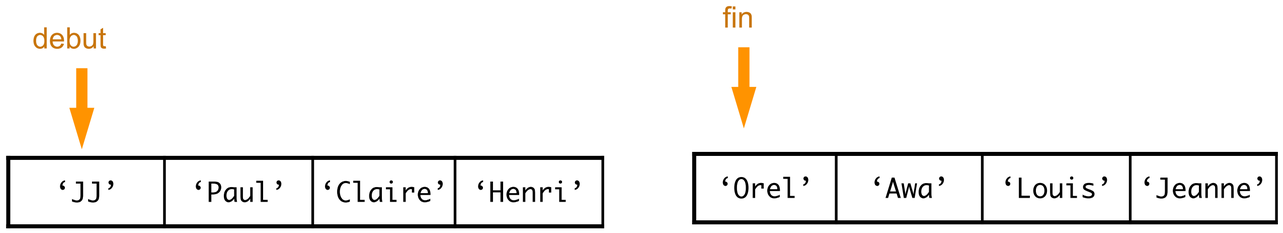
Signature d'un module

- le résultat de la déclaration d'un module

```
(* module Fifo1 :
  sig
    type 'a queue = { debut : 'a list; fin : 'a list; }
    val make : 'a list -> 'a list -> 'a queue
    val empty : 'a queue
    val is_empty : 'a queue -> bool
    val add : 'a -> 'a queue -> 'a queue
    exception Empty
    val fst : 'a queue -> 'a
    val pop : 'a queue -> 'a queue
  end
*)
```

- la signature est le type du module





les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

Signature d'un module

- on peut déclarer une signature plus abstraite

```
module type FIFO =
    sig
      type 'a queue
      val empty : 'a queue
      val add : 'a -> 'a queue -> 'a queue
      val fst : 'a queue -> 'a
      val pop : 'a queue -> 'a queue
      exception Empty
    end;;
```

- le module devient plus **abstrait** (ici `make` n'est plus public)

```
module Fifo = (Fifo1 : FIFO) ;;
```

- l'utilisateur du module `Fifo` ne connait pas son implémentation

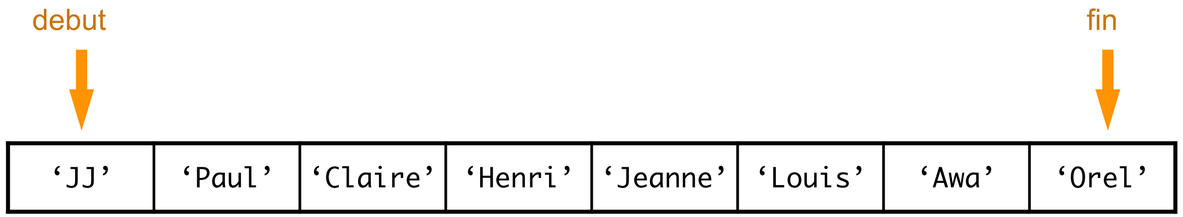
```
let q = Fifo.empty ;;
let q1 = Fifo.add "JJ" q ;;
let q2 = Fifo.add "Paul" q1 ;;
let q3 = Fifo.add "Claire" q2 ;;
let q4 = Fifo.pop q3 ;;
Fifo.fst q3 ;;
(* - : string = "JJ" *)
Fifo.fst q4 ;;
(* - : string = "Paul" *)
```

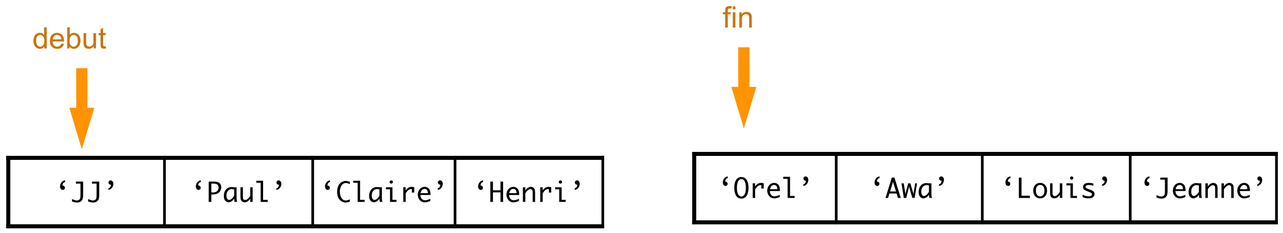
les sources sont en `http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py`

Signature d'un module

- autre implémentation avec des paires de listes

```
module Fifo2 =
    struct
      type 'a queue = 'a list * 'a list
      let make debut fin =
        match debut with
        | [ ] -> (List.rev fin), [ ]
        | _  -> debut, fin
      let empty = [ ], [ ]
      let is_empty = function ([ ], _ ) -> true | _ -> false
      let add x (debut, fin) = make debut (x :: fin)
      exception Empty
      let fst = function
        | [ ], _ -> raise Empty
        | (x :: _), _  -> x
      let pop = function
        | [ ], _  -> raise Empty
        | (_:: d), f -> make d f
    end;;
```





- on abstrait le module avec la même signature abstraite

```
module Fifo = (Fifo2 : FIFO) ;;
```

les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

Signature d'un module

- syntaxes alternatives

```
module Fifo = (struct ... end : FIFO);;

module Fifo : FIFO = struct ... end;;

module Fifo = (struct ... end : sig ... end) ;;

module Fifo : sig ... end = struct ... end ;;
```

les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

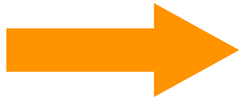
Foncteurs

- paramétrisation d'un module

```
type comparison = Less | Equal | Greater;;

module type ORDERED =
    sig
      type t
      val compare: t -> t -> comparison
    end;;

module SetList =
    functor (Elt: ORDERED) ->
      struct
        type element = Elt.t
        type set = element list
        let empty = [ ]
        let rec add x s =
          match s with [ ] -> [x]  | y::s' ->
             match Elt.compare x y with
             | Equal   -> s
             | Less    -> x :: s
             | Greater -> y :: add x s'
        let rec member x s =
          match s with [ ] -> false | y::s' ->
              match Elt.compare x y with
              | Equal   -> true
              | Less    -> false
              | Greater -> member x s'
      end;;
```



```
(* module type ORDERED =
  sig
    type t
    val compare : t -> t -> comparison
  end
*)


(* module SetList :
  functor (Elt : ORDERED) ->
   sig
     type element = Elt.t
     type set = element list
     val empty : 'a list
     val add : Elt.t -> Elt.t list -> Elt.t list
     val member : Elt.t -> Elt.t list -> bool
   end
*)
```

les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

Foncteurs

- instanciation d'un module

```
module OrderedString =
    struct
      type t = string
      let compare x y = if x = y then Equal
         else if x < y then Less else Greater
    end;;


module StringSet = SetList (OrderedString);;


let s = StringSet.empty ;;
let s1 = StringSet.add "JJ" s ;;
let s2 = StringSet.add "Paul" s1 ;;
let s3 = StringSet.add "Claire" s2 ;;
StringSet.member "Paul" s3 ;;
```

```
(* module StringSet :
  sig
    type element = OrderedString.t
    type set = element list
    val empty : 'a list
    val add : OrderedString.t -> OrderedString.t list -> OrderedString.t list
    val member : OrderedString.t -> OrderedString.t list -> bool
  end
*)
```

les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

Foncteurs

- instanciation d'un module

```
module Num =
  struct
    type t = I of int | F of float
    let compare x y =
      let float_of_num = function I x -> float_of_int x | F x -> x in
      let x' = float_of_num x in
      let y' = float_of_num y in
      if x' = y' then Equal else if x' < y' then Less else Greater
  end ;;


module NumSet = SetList (Num) ;;


let s = NumSet.empty ;;
let s1 = NumSet.add (I 3) s ;;
let s2 = NumSet.add (F 1.5) s1 ;;
let s3 = NumSet.add (I 4) s2 ;;
NumSet.member (F 4.0) s3 ;;
```

```
(* module NumSet :
 sig
   type element = OrderedNum.t
   type set = element list
   val empty : 'a list
   val add : OrderedNum.t -> OrderedNum.t list -> OrderedNum.t list
   val member : OrderedNum.t -> OrderedNum.t list -> bool
 end
*)
```

les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

Foncteurs

- au lieu de SetList on peut aussi faire un module Set2 avec des arbres binaires de recherche
- on peut abstraire la représentation des ensembles en précisant la signature du foncteur

```
module type SETFUNCTOR =
    functor (Elt: ORDERED) ->
      sig
        type element = Elt.t
        type set
        val empty : set
        val add : element -> set -> set
        val member : element -> set -> bool
      end;;

module Set = (SetList : SETFUNCTOR);;


module StringSet = Set (OrderedString);;


let s = StringSet.empty ;;
let s1 = StringSet.add "JJ" s ;;
let s2 = StringSet.add "Paul" s1 ;;
StringSet.member "Paul" s2 ;;
(* - : bool = true *)
s ;;
(* - : StringSet.set = <abstr> *)
```

```
(* module StringSet :
 sig
   type element = OrderedString.t
   type set = Set(OrderedString).set
   val empty : set
   val add : element -> set -> set
   val member : element -> set -> bool
 end
*)
```

les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

Foncteurs

- écriture plus élégante en définissant un type de module abstrait

```
module type SET =
    sig
      type element
      type set
      val empty : set
      val add : element -> set -> set
      val member : element -> set -> bool
    end;;


module Set =
    (SetList : functor(Elt: ORDERED) -> (SET with type element = Elt.t));;
```

```
(* module Set :
  functor (Elt : ORDERED) ->
   sig
     type element = Elt.t
     type set
     val empty : set
     val add : element -> set -> set
     val member : element -> set -> bool
   end
*)
```

- écritures alternatives comme pour les modules

```
module Set: functor (Elt: ORDERED) -> (SET with type element = Elt.t) =
    SetList ;


module Set (Elt: ORDERED) : (SET with type element = Elt.t) =
    struct .. end ;;
```

les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

Foncteurs

- écriture plus élégante en définissant un type de module abstrait

```
module Set (Elt: ORDERED) : (SET with type element = Elt.t) =
  struct
    type element = Elt.t
    type set = element list
    let empty = [ ]
    let rec add x s =
      match s with [ ] -> [x]  | y::s' ->
         match Elt.compare x y with
         | Equal   -> s
         | Less    -> x :: s
         | Greater -> y :: add x s'
    let rec member x s =
      match s with [ ] -> false | y::s' ->
          match Elt.compare x y with
          | Equal   -> true
          | Less    -> false
          | Greater -> member x s'
  end ;;
```

```
(* module Set :
  functor (Elt : ORDERED) ->
   sig
     type element = Elt.t
     type set
     val empty : set
     val add : element -> set -> set
     val member : element -> set -> bool
   end
*)
```

les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

Modules

- en bref :

```
module type SET =
    sig
      type element
      type set
      val empty : set
      val add : element -> set -> set
      val member : element -> set -> bool
    end;;
```

```
type comparison = Less | Equal | Greater;;

module type ORDERED =
    sig
      type t
      val compare: t -> t -> comparison
    end;;
```

```
module Set (Elt: ORDERED) : (SET with type element = Elt.t) =
  struct
    type element = Elt.t
    type set = element list
    let empty = [ ]
    let rec add x s =
      match s with [ ] -> [x]  | y::s' ->
         match Elt.compare x y with
         | Equal   -> s
         | Less    -> x :: s
         | Greater -> y :: add x s'
    let rec member x s =
      match s with [ ] -> false | y::s' ->
          match Elt.compare x y with
          | Equal   -> true
          | Less    -> false
          | Greater -> member x s'
  end ;;
```

```
module OrderedString =
    struct
      type t = string
      let compare x y = if x = y then Equal
          else if x < y then Less else Greater
    end;;

module StringSet = Set (OrderedString);;

let s = StringSet.empty ;;
let s1 = StringSet.add "JJ" s ;;
let s2 = StringSet.add "Paul" s1 ;;
StringSet.member "Paul" s2 ;;
(* - : bool = true *)
s ;;
(* - : StringSet.set = <abstr> *)
```

les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

Modules

- un autre module ne tenant pas compte des majuscules ou minuscules

```
module OrderedNoCaseString =
  struct
    type t = string
    let compare x y =
      let x' = String.lowercase_ascii x in
      let y' = String.lowercase_ascii y in
      if x' = y' then Equal else if x' < y' then Less else Greater
  end;;


module NoCaseStringSet = Set (OrderedNoCaseString) ;;

let ns = NoCaseStringSet.empty ;;
let ns1 = NoCaseStringSet.add "JJ" ns ;;
let ns2 = NoCaseStringSet.add "Paul" ns1 ;;
StringSet.member "PAUL" s2 ;;
(* - : bool = true *)
ns ;;
(* - : NoCaseStringSet.set = <abstr> *)
```

Exercice Faire des ensembles d'entiers, de flottants, de nums, d'arbres.

les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

Compilation

- un programme peut être compilé avec la commande `ocamlc`

```
ocamlc -o prog prog.ml
```

prog.ml → prog (exécutable)

`ocamlc prog.ml` produit `a.out` (exécutable)

- un programme peut être composé de plusieurs unités de compilation

```
ocamlc -o prog prog1.ml prog2.ml
```

- une variable d'une unité de compilation précédente est référencée avec la notation qualifiée

dans prog1.ml `let f x = ...`

dans prog2.ml `if Prog1.f x > 10 then ..` (ZZ: nom commençant par une majuscule, le nom du fichier par une minuscule)

- on peut aussi ouvrir prog1 dans prog2 avec `open Prog1`

les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

Compilation séparée

- la compilation peut être séparée en plusieurs étapes

`ocamlc -c prog1.ml` `prog1.ml` → `prog1.cmo` (code objet) + `prog1.cmi` (interface)

`ocamlc -c prog2.ml` `prog2.ml` → `prog2.cmo` (code objet) + `prog2.cmi` (interface)

- on finit par l'édition de liens (*link*)

`ocamlc -o prog prog1.cmo prog2.cmo` → `prog` (exécutable)

- les fichiers d'interfaces servent à retrouver les types des variables exportées

- les fichiers d'interfaces non compilés peuvent servir à la documentation

`ocamlc -i prog.ml > prog.mli` → `prog.mli` (interface lisible)

les sources sont en `http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py`

Modules et compilation séparée

mylib.ml

```
type comparison = Less | Equal | Greater;;

module type ORDERED =
    sig
      type t
      val compare: t -> t -> comparison
    end;;

module type SET =
    sig
      type element
      type set
      val empty : set
      val add : element -> set -> set
      val member : element -> set -> bool
    end;;

module Set (Elt: ORDERED) : (SET with type element = Elt.t) =
  struct
    type element = Elt.t
    type set = element list
    let empty = [ ]
    let rec add x s =
      match s with [ ] -> [x]  | y::s' ->
         match Elt.compare x y with
         | Equal   -> s
         | Less    -> x :: s
         | Greater -> y :: add x s'
    let rec member x s =
      match s with [ ] -> false | y::s' ->
          match Elt.compare x y with
          | Equal   -> true
          | Less    -> false
          | Greater -> member x s'
  end ;;
```

mylib_ext.ml

```
open Mylib;;

module OrderedString =
    struct
      type t = string
      let compare x y = if x = y then Equal
         else if x < y then Less else Greater
    end;;

module StringSet = Set (OrderedString);;
```

main.ml

```
open Mylib_ext ;;

let s = StringSet.empty ;;
let s1 = StringSet.add "JJ" s ;;
let s2 = StringSet.add "Paul" s1 ;;
Printf.printf "%b\n" (StringSet.member "Paul" s2) ;;
```

les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

Signatures et compilation séparée

mylib.mli

```
type comparison = Less | Equal | Greater
module type ORDERED = sig type t val compare : t -> t -> comparison end
module type SET =
  sig
    type element
    type set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end
module Set :
  functor (Elt : ORDERED) ->
    sig
      type element = Elt.t
      type set
      val empty : set
      val add : element -> set -> set
      val member : element -> set -> bool
    end
```

stringset.mli

```
module OrderedString :
  sig type t = string val compare : 'a -> 'a -> Mylib.comparison end
module StringSet :
  sig
    type element = OrderedString.t
    type set = Mylib.Set(OrderedString).set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end
```

les sources sont en http://jeanjacqueslevy.net/prog-py-22/progs/c8a.py

Conclusion

VU:

- modules
- signatures
- foncteurs
- exemples de modules paramétrés
- compilation séparée

TODO list

- objets
- algorithmique
- parallélisme