MdP : COURS 3

Modèles de Programmation

Emmanuel Chailloux

PLAN

- programmation impérative
  - Entrées/sorties
  - valeurs physiquement modifiables
  - structures de contrôle
- éléments de choix du style (fonctionnel ou impératif)
- représentation des fermetures
- évaluation retardée et streams

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, ...

Entrées/sorties

Canaux:

- types : *in_channel* et *out_channel*
- fonctions : `open_in` : *string*$\rightarrow$ *in_channel* (`close_in`)
  `open_out` : *string*$\rightarrow$*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* → *string* → *int* → *int* → *int*
`input_line`	:	*in_channel* → *string*
`output`	:	*out_channel* → *string* → *int* → *int* → *unit*
`output_string`	:	*out_channel* → *string* → *unit*
`read_line`	:	*unit* → *string*
`read_int`	:	*unit* → *int*
`print_string`	:	*string* → *unit*
`print_int`	:	*int* → *unit*
`print_newline`	:	*unit* → *unit*

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 à champs modifiables, chaînes de caractères, références

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

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

Vecteurs (1)

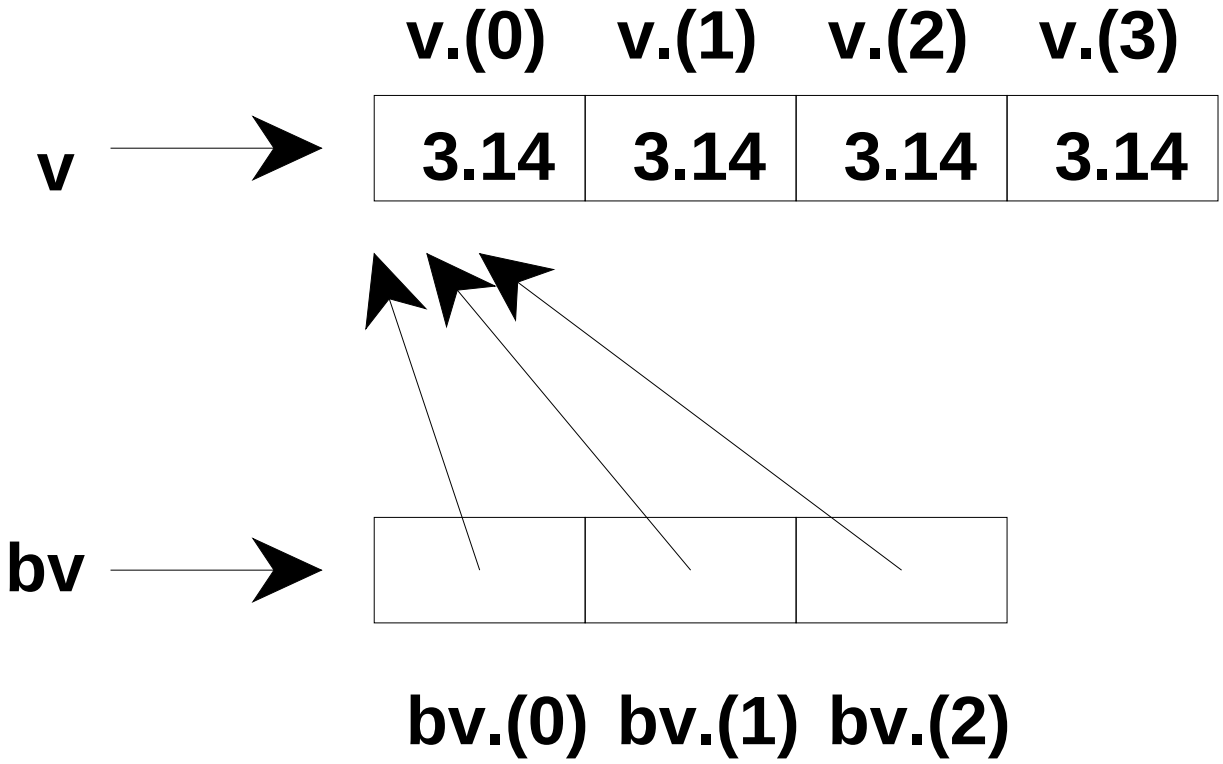
- regroupent un nombre connu d'éléments de même type
- création : `Array.create` : $int \rightarrow$ *'a* $\rightarrow$ *'a array*,
- longueur : `Array.length` : *'a array*$\rightarrow$ *int*
- accès : $e_1.(e_2)$
- modification : $e_1.(e_2)$<-e_3

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|]
```

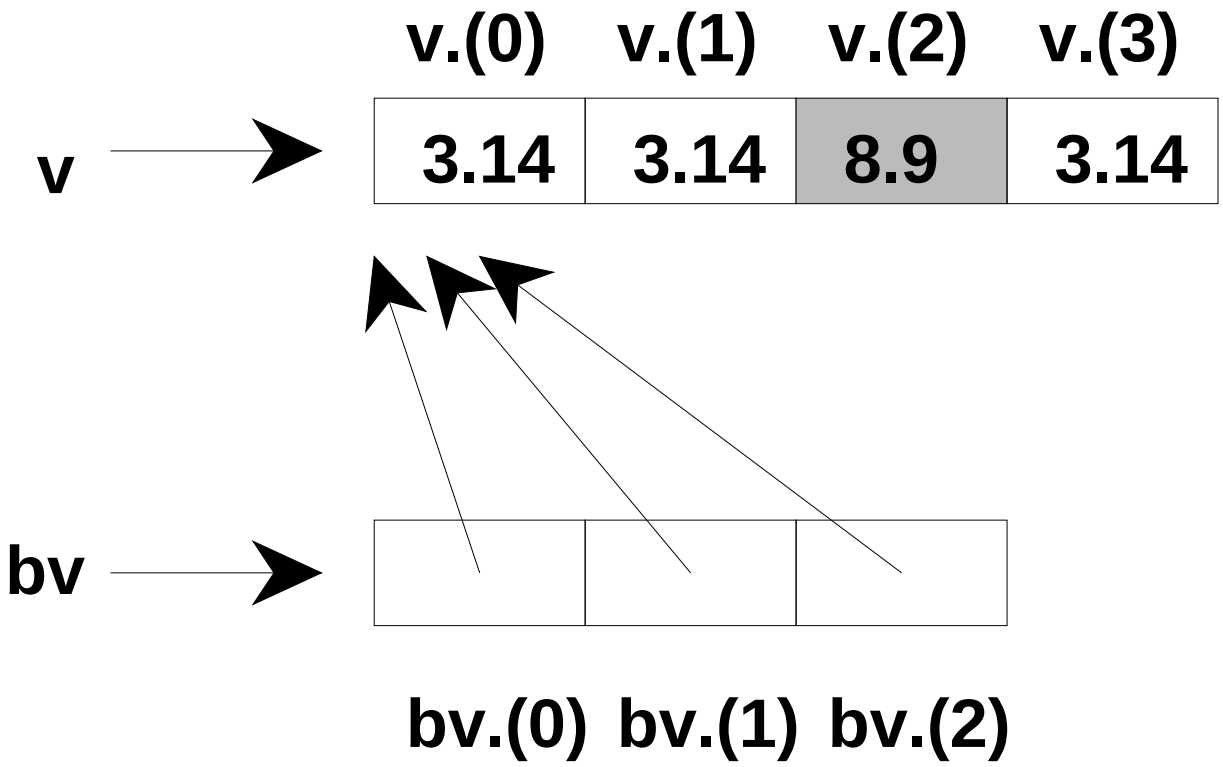
Représentation mémoire (1)

```
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` : $int \rightarrow int \rightarrow 'a \rightarrow 'a\ array\ array$
- itérateurs
  - `iter` : $('a \rightarrow unit) \rightarrow 'a\ array \rightarrow unit$
  - `map` : $('a \rightarrow 'b) \rightarrow 'a\ array \rightarrow 'b\ array$
  - `iteri` : $(int \rightarrow 'a \rightarrow unit) \rightarrow 'a\ array \rightarrow unit$
  - `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}
```

Chaînes de caractères

- les chaînes sont des valeurs modifiables (fonction `input`)
- accès : $e_1.[e_2]$
- modification : $e_1.[e_2]$<-e_3

```
1  # let s  = "bonjour";;
2  val s : string = "bonjour"
3  # s.[3];;
4  - : char = 'j'
5  # s.[3]<-'-';;
6  - : unit = ()
7  # s;;
8  - : string = "bon-our"
```

Références

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

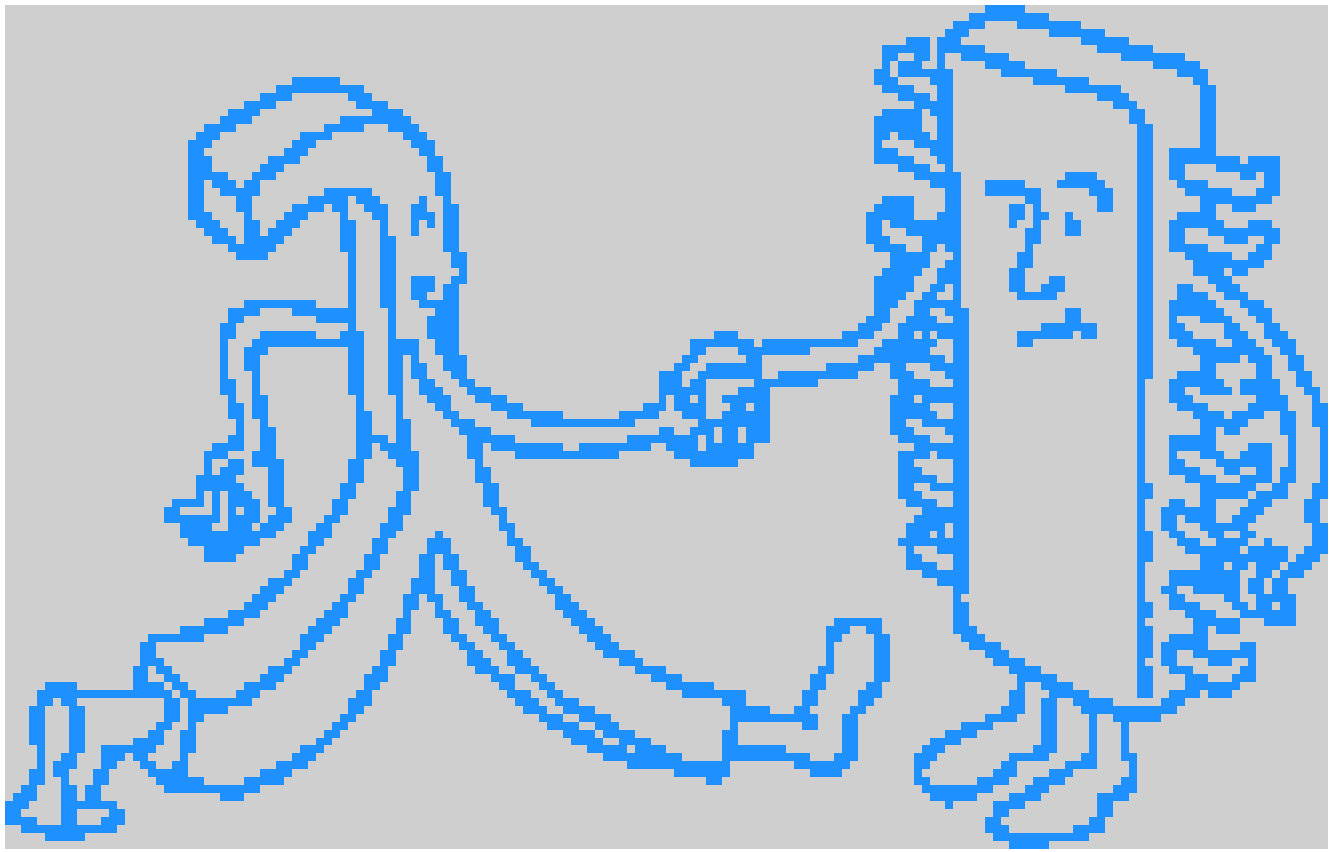
- séquentielle : $e1$; e_2; ...; e_n
  regroupée : (...) ou **begin** ... **end**
  le type de la séquence est le type de e_n
- conditionnelle : **if** c_1 **then** e_2 (e_2 de type *unit*)
- itératives :
  - **while** c **do** e **done**
  - **for** v=e_1 [**down**]**to** e_2 **do** e_3 **done**

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
  - valeurs fonctionnelles modifiables
  - implantation de l'évaluation retardée

Fonction : map

- style fonctionnel

```
# let rec fmap f l = match l with
  [] -> []
| h::t -> let r = f h in r::(fmap f t);;
val fmap : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- style impératif

```
# let imap f l  =
  let nl = ref l
  and nr = ref [] in
    while (!nl <> []) do
      nr := ( f (List.hd !nl)) :: (!nr);
      nl := List.tl !nl
    done;
    List.rev !nr;;
val imap : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Transposée de matrice

- style impératif

```
# let itrans m = let l = Array.length m in
    for i=0 to l-1 do
      for j=i to l-1 do
        let v = m.(i).(j) in
          m.(i).(j) <- m.(j).(i);
          m.(j).(i) <- v
      done done;;
val itrans : 'a array array -> unit = <fun>
```

- style fonctionnel

```
# let rec  ftransl l = match l with
  []::_ -> []
| _ ->   (List.map List.hd l) ::
         ftransl (List.map List.tl l);;
val ftransl : 'a list list -> 'a list list = <fun>
```

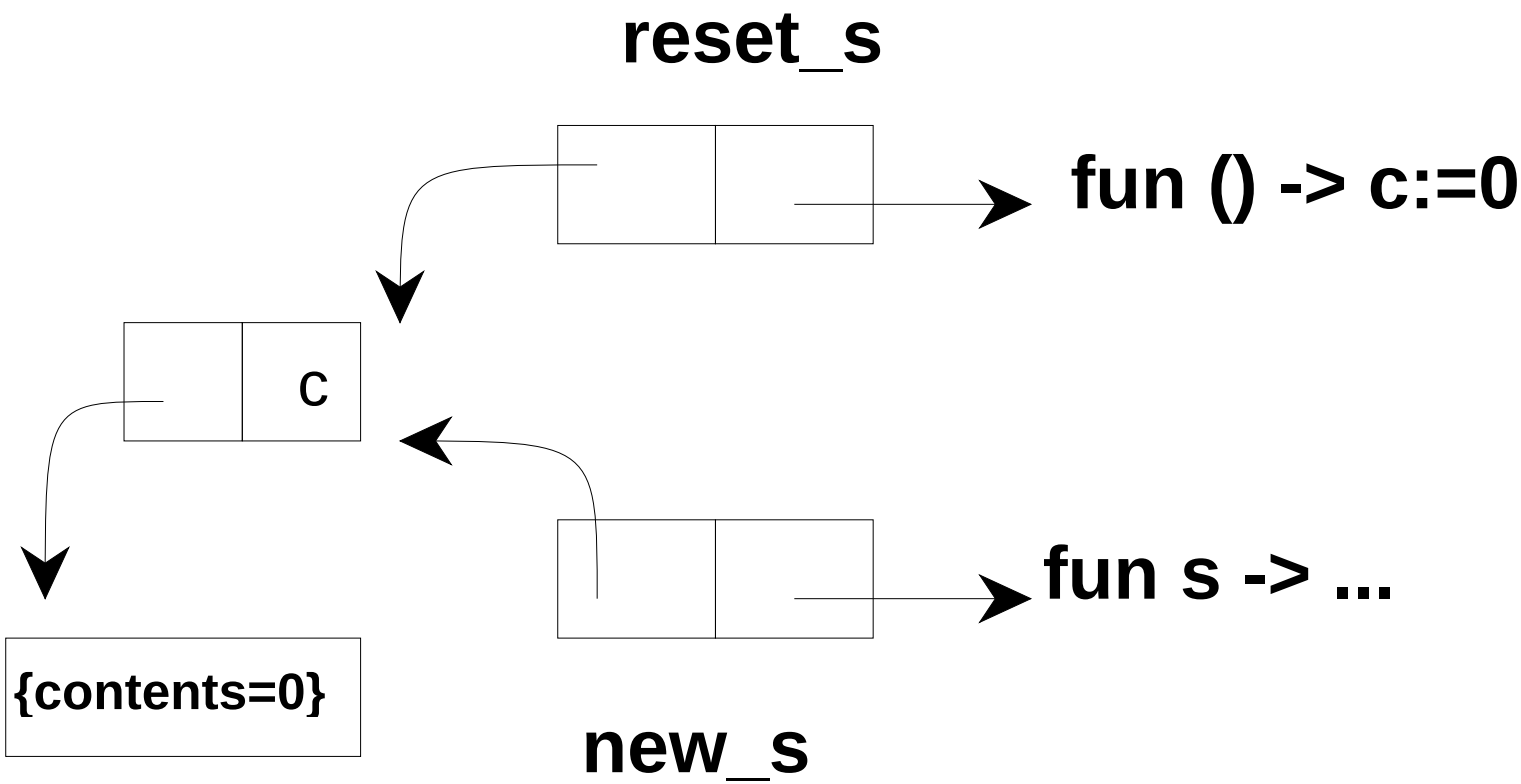
Représentation des fermetures

- 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



Listes codées par des fermetures (1)

```
# exception Empty_list;;
exception Empty_list
# let nil = function x -> raise Empty_list;;
val nil : 'a -> 'b = <fun>
# let lpm = ref nil;;
val lpm : ('_a -> '_b) ref = {contents=<fun>}
# let cons a l =
    function e ->
      print_int a; print_string "-"; print_int e;
      if e = a then true else l e;;
val cons : int -> (int -> bool) -> int -> bool = <fun>
#   let mcons a l = l := cons a (!l);;
val mcons : int -> (int -> bool) ref -> unit = <fun>
# mcons 3 lpm;;
- : unit = ()
# lpm;;
- : (int -> bool) ref = {contents=<fun>}
# mcons 8 lpm;;
- : unit = ()
```

Listes codées par des fermetures (2)

```
# let mem a l =
  try !l a
  with Empty_list -> false;;
val mem : 'a -> ('a -> bool) ref -> bool = <fun>
# mem 3 lpm;;
8-3 3-3
- : bool = true
# mem 9 lpm;;
8-9 3-9
- : bool = false
```

Evaluation retardée

calcul gelé dans une fermeture :

```
1  # type 'a v =
2      Imm of 'a
3    | Ret of (unit -> 'a);;
4
5  # type 'a vm = {mutable c : 'a v };;
6
7  # let  eval e = match e.c with
8    Imm a -> a
9  | Ret f -> let u = f () in
10                ( e.c <- Imm u;u);;
11 val eval : 'a vm -> 'a = <fun>
```

If fonctionnel retardé

```
1  # let si_ret c e1 e2 =
2    if eval c then eval e1 else eval e2;;
3
4  val si_ret : bool vm -> 'a vm -> 'a vm -> 'a = <fun>
5
6  # let rec facr n =
7    si_ret {c=Ret(fun () -> n = 0)}
8           {c=Ret(fun () ->  1)}
9           {c=Ret(fun () -> n*(facr(n-1)))};;
10
11 val facr : int -> int = <fun>
12
13 # facr 5;;
14 - : int = 120
```

Module Lazy

```
1  # let u = lazy (print_string "trace";
2                   print_newline(); 4*5);;
3  val u : int Lazy.status ref = {contents=Lazy.Delayed <fun↩
       >}
4  # Lazy.force u;;
5  trace
6  - : int = 20
7  # Lazy.force u;;
8  - : int = 20
```

L'évaluation retardée permet de manipuler des données potentiellement infinies!!!

Structures paresseuses modifiables

```
1  # type 'a ens = {mutable i : 'a; f :'a -> 'a};;
2  type 'a ens = { mutable i: 'a; f: 'a -> 'a }
3  #   let next e = let x = e.i in e.i<-e.f e.i; x;;
4  val next : 'a ens -> 'a = <fun>
5
6  # let nat = {i=0; f=fun x -> x + 1};;
7  val nat : int ens = {i=0; f=<fun>}
8  # [next nat; next nat; next nat];;
9  - : int list = [2; 1; 0]
10
11 # let fib = {i=1; f= let c = ref 0 in
12                      fun v -> let r = !c + v in c:=v;r};;
13 val fib : int ens = {i=1; f=<fun>}
14 # [next fib;next fib; next fib; next fib];;
15 - : int list = [3; 2; 1; 1]
```

Streams

- flots ou flux
- séquence, d'éléments de même type
- potentiellement infinie
- type abstrait
- filtrage sur les *streams*
- utilisée pour les analyses lexicale et syntaxique
- analyseur descendant (prédictif)
- sans contexte ou avec contexte
- extension du langage O'CAML

Construction de streams

```
1  $ ocaml
2           Objective Caml version 3.10.0
3
4  # #load "camlp4o.cma";;
5           Camlp4 Parsing version 3.10.0
6
7  #
8
9  # [<>];;
10 - : 'a Stream.t = <abstr>
11 # [< '0; '2; '4 >];;
12 - : int Stream.t = <abstr>
13 # let s = [< '1; '3>] in [< s;'8>];;
14 - : int Stream.t = <abstr>
```

Les éléments non *quotés* sont vus comme des sous-streams.

Fonctions prédéfinies

`Stream.of_channel`	:	*in_channel* $\rightarrow$ *char Stream.t*
`Stream.of_string`	:	*string* $\rightarrow$ *char Stream.t*
`Stream.of_list`	:	*'a list* $\rightarrow$ *'a Stream.t*

Filtrage de streams

Syntaxe:

match e **with parser** p_1 -> e_1 | ... | p_n -> e_n

```
1
2  # let next s =
3    match s with parser
4        [< 'x >] -> x
5    |   [< >] -> raise (Failure "stream vide");;
6         val next : 'a Stream.t -> 'a = <fun>
7  # let s = [< '0; '1; >];;
8  val s : int Stream.t = <abstr>
9  # next s;;
10 - : int = 0
11 # next s;;
12 - : int = 1
```

Accès destructif

```
1  # next s;;
2  Uncaught exception: Failure("stream vide")
3  # let rec somme s =
4    match s with parser
5     [<'x ; p = somme  >] -> x+p
6   | [<>] -> 0;;
7  val somme : int Stream.t -> int = <fun>
8  # let x = [< '1; '2; '3; '4 >];;
9  val x : int Stream.t = <abstr>
10 # somme x;;
11 - : int = 10
12 # somme x;;
13 - : int = 0
```

Résumé des expressions

```
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          | [< 'e1; 'e2; e3; ...; en >]
8          | lazy expr
9          | match expr with parse filtrage
```