Aspects impératifs du langage Caml

Solange Coupet Grimal

Cours Logique, Déduction, Programmation 2002–2004*

Nous avons jusqu'à présent décrit la partie purement fonctionnelle du langage Caml. Nous avons vu en particulier que l'évaluation d'une expression purement fonctionnelle donne toujours le même résultat quelle que soit la stratégie (par nom ou par valeur) adoptée, à condition toutefois que chacune des évaluations termine.

Cependant, pour des raisons pratiques et pour que le langage soit réellement efficace, l'introduction de certains traits « impurs »est incontournable. Les fonctions qui interagissent avec le système (entrées/sorties, fonction `exit`), les exceptions (cf. la fonction `failwith` qui provoque entre autre l'abandon des calculs en cours), les opérations destructives (affectation) qui font référence à la représentation en machine des structures de données ne font pas partie du noyau fonctionnel du langage. Elles peuvent être vues comme des « effets de bord »de l'évaluation d'une expression et en ce sens elles dépendent de la stratégie adoptée.

Ce qui suit a pour objet d'introduire et de préciser ces aspects.

1 Exceptions

1.1 Le type `exn`

Caml comporte un type avec constructeurs prédéfini, le type : `exn`.

```
type exn = ...
           Division_by_zero
         | Failure of string

           ...;;

#Division_by_zero;;
- : exn = Division_by_zero
```

*Texte révisé par Luigi Santocanale le 24 octobre 2004.

```
#Failure "zut";;
- : exn = Failure "zut"
```

1.2 Génération d'une exception

Il existe une fonction prédéfinie sur le type `exn` qui permet de générer une exception (on dit « lever ») une exception.

```
#raise;;
- : exn -> 'a = <fun>
```

Par exemple :

```
# let hd = function  [] -> raise (Failure "hd")
  |  (a::l)    -> a;;

#hd [];;
Uncaught exception: Failure "hd"
```

Ces exceptions apparaissent donc en général sous la forme d'un message donnant le nom de l'exception et son argument.

```
#1/0;;
Uncaught exception: Division_by_zero
```

Cette fonction `raise` n'est pas définissable dans le langage lui-même et a pour effet d'interrompre tous les calculs en cours pour déclencher la valeur exceptionnelle qu'elle a reçue en argument. Elle est polymorphe en ce sens que le type `'a` de la valeur retournée sera évalué de façon à ce qu'il soit compatible avec le contexte à condition que ce soit possible.

```
#1 + (raise (Failure "zut"));;
Uncaught exception: Failure "zut"

#"hello"^ (raise (Failure "zut"));;
Uncaught exception: Failure "zut"

#1 + (raise (Failure "zut"))^"hello";;
Toplevel input:
>1 + (raise (Failure "zut"))^"hello";;
>^^^^^^^^^^^^^^^^^^^^^^^^^^
This expression has type int,
but is used with type string.
```

L'exception `Failure s` – où `s` est un message d'erreur et donc une chaîne de caractères – étant très utilisée, on a prédéfini une fonction `failwith` de la façon suivante :

```
#let failwith s = raise (Failure s);;
failwith : string -> 'a = <fun>
```

D'où par exemple la définition de la fonction `hd` :

```
# let hd = function  [] -> failwith "hd"
  | (a::l) -> a;;
```

1.3 Extension du type `exn`

Le type `exn`, bien que prédéfini, est extensible. Contrairement aux autres types, on peut rajouter de nouveaux constructeurs de la façon suivante :

```
exception <nom du constructeur> of <type de l'argument>
```

Exemple :

```
#exception Int_exception of int;;
Exception Int_exception defined.

#let pred n = if n<=0 then (raise (Int_exception n))
                 else n-1;;
pred : int -> int = <fun>

#pred 2;;
- : int = 1

#pred (-3);;
Uncaught exception: Int_exception -3
```

1.4 Récupération d'une erreur

Il existe une possibilité de « récupérer »les exceptions déclenchées par la fonction `raise`. L'exception récupérée est remplacée par une certaine valeur et l'évaluation en cours se poursuit normalement.

Pour cela on utilise la construction

```
try ... with ...
```

Plus précisément considérons l'expression :

```
try e with
    p1 -> e1
  | p2 -> e2
      ...
  | pn -> en;;
```

Si `e` est évaluée normalement (le résultat n'est pas une exception), alors la valeur de cette expression est la valeur de `e`. Si `e` est évaluée en une exception filtrée par `pi`, alors la valeur de l'expression est celle de `ei`.

Exemple. Avec la définition de `pred` ci-dessus :

```
#let f n = try (pred n) with (Int_exception 0) -> 0;;
f : int -> int = <fun>

#f 3;;
- : int = 2

#f 0;;
- : int = 0

#f (-1);;
Uncaught exception: Int_exception -1
```

2 Entrées/Sorties

Rappelons que le type `unit` n'a qu'une seule valeur notée `()`.

2.1 Fonctions de sorties

Entiers : la fonction `print_int` prend en argument une expression de type entier et renvoie `()`. L'évaluation d'une expression `print_int e` a pour effet de bord l'impression à l'écran de la valeur de `e`.

```
#print_int;;
- : int -> unit = <fun>

#print_int 5;;
5- : unit = ()
```

```
#print_int (5+6*2);;
17- : unit = ()

#print_int (2/0);;
Uncaught exception: Division_by_zero
```

La fonction analogue pour les chaînes est la fonction `print_string`.

```
#print_string;;
- : string -> unit = <fun>

#print_string "hello";;
hello- : unit = ()

#print_string (string_of_float ((5.1+.8.2)/.3.));;
4.43333333333- : unit = ()
```

Cette dernière expression a le même effet que celle utilisant la fonction d'impression des flottants `print_float` :

```
#print_float ((5.1+.8.2)/.3.);;
4.43333333333- : unit = ()
```

La session ci-dessous décrit d'autres fonctions de sortie et des exemples d'utilisation.

```
#print_newline;;
- : unit -> unit = <fun>

#print_newline();;

- : unit = ()

#print_char;;
- : char -> unit = <fun>

#print_char 'a';;
a- : unit = ()

#print_char '\n';;

- : unit = ()
```

```
#print_string "\n Il y a dans les bois \n Des arbres fous d'oiseaux \n\n";;

 Il y a dans les bois
 Des arbres fous d'oiseaux

- : unit = ()

#print_endline;;
- : string -> unit = <fun>

#print_endline "\n Il y a dans les bois \n Des arbres fous d'oiseaux";;

 Il y a dans les bois
 Des arbres fous d'oiseaux
- : unit = ()
```

2.2 Fonctions de lecture

La fonction `read_int` a pour argument `()` et renvoie un entier lu sur le canal d'entrée standard (clavier).

```
#read_int;;
- : unit -> int = <fun>

#read_int();;
9
- : int = 9

#let x = read_int();;
89
x : int = 89

#let x = fact (read_int());;
12
x : int = 479001600

#read_int();;
5 6;;
Uncaught exception: Failure "int_of_string"
```

Dans la dernière expression, la chaîne de caractères `"5 6"` n'a pu être convertie en un entier. La session suivante traite de la lecture des flottants et des chaînes de caractères et se passe de commentaires.

```
#read_float;;
- : unit -> float = <fun>

#read_float();;
34.88;;
- : float = 34.88

#read_float();;
67e3;;
- : float = 67000.0

#read_float();;
-34E-1
- : float = -3.4

#read_line;;
- : unit -> string = <fun>

# let s = read_line();;
Il y a dans le bois
s : string = "Il y a dans le bois"
```

3 Séquencement

Si `e1` et `e2` sont 2 expressions, le séquencement de `e1` et `e2` est noté :

```
(e1;e2)
```

ou encore :

```
begin e1;e2 end
```

La valeur de `(e1 ;e2)` est la valeur de `e2` si l'évaluation de `e1` et `e2` ne comporte pas d'exception.

Si l'évaluation de `e1` provoque une exception `ex1` alors celle de `(e1 ;e2)` aussi, sinon si `e1` s'évalue normalement et que `e2` s'évalue en une exception `ex2`, alors `(e1 ; e2)` s'évalue en `ex2`.

Signalons la fonction `List.iter` qui est telle que :

```
List.iter f [a1;a2;...;an] = begin (f a1); (f a2) ; .... (f an);() end;;

#List.iter;;
- : ('a -> 'b) -> 'a list -> unit = <fun>
```

Exemple.

Impression des éléments d'une liste d'entiers.

```
#let print_list = fun l -> (List.iter print_int l ; print_newline());;

print_list : int list -> unit = <fun>

#print_list ([4;6;8]@[1;2]);;
46812
- : unit = ()
```

Examiner la session suivante.

```
#(List.map print_int [1;2;3]);;
123- : unit list = [(); (); ()]

# (List.map print_int [1;2;3]; print_newline());;
Characters 1-29:
Warning: this expression should have type unit.
  (List.map print_int [1;2;3]; print_newline());;
   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
123
- : unit = ()

# (List.iter print_int [1;2;3]; print_newline());;
123
- : unit = ()
```

Définissons maintenant

```
#let map_right f l  = List.fold_right (fun a l -> (f a)::l) l [];;
map_right : ('a -> 'b) -> 'a list -> 'b list = <fun>

#map_right print_int [1;2;3];;
321- : unit list = [(); (); ()]
```

Résultats d'examen. La fonction `admission` suivante prend en argument une fonction `f` sur 2 flottants et renvoie la pondération `f e o` d'une note d'écrit `e` et d'une note d'oral `o` entrées au clavier sous forme de flottants.

```
#let admission  = fun f ->      print_string "\nnote d'ecrit: ";
                                let e = read_float() in
                                        (print_string "\nnote d'oral: ";
                                                let o = read_float() in
                                                        (f e o));;
admission : (float -> float -> 'a) -> 'a = <fun>
```

On remarque que l'évaluation de l'expression ci-dessus n'a provoqué aucune impression de chaînes de caractères à l'écran puisque les appels à la fonction `print_string` interviennent sous un `fun`, autrement dit dans une abstraction.

En revanche, si on instancie le parametre `f` en appliquant `admission` a une fonction donnée comme ci-dessous, l'évaluation provoque les lectures et les affichages à l'écran :

```
#admission  (fun x y -> (x +. y)/.2.);;

note d'ecrit: 14.5

note d'oral: 12.1
- : float = 13.3
```

On peut imaginer que pour une matière donnée, la fonction de pondération est fixée. On décide de la définir une fois pour toutes pour ne pas avoir à la récrire à chaque calcul. De plus cette fonction calculera non seulement la moyenne pondérée des notes d'écrit et d'oral mais aussi affichera à l'écran la chaîne `admis` ou `refuse` selon que cette moyenne est supérieure ou strictement inférieure à 10.

```
#let ponderation = (fun a b -> let r = (3.*.a+.b)/. 4. in
                        print_string (if r<10. then "ajourne" else "admis");
                        print_newline();
                        print_newline());;
ponderation : float -> float -> unit = <fun>
```

Comme nous l'avons remarque, l'évaluation de l'expression :

```
(admission ponderation)
```

va provoquer les affichages et les lectures. En effet,

```
admission  ponderation =( (fun f -> print_string "\nnote d'ecrit: ";
                                let e = read_float() in
                                        (print_string "\nnote d'oral: ";
                                                let o = read_float() in
                                                        (f e o)))

                          (fun a b -> let r= (3.*.a+.b)/. 4. in
                          print_string (if r<10. then "ajourne" else "admis");
                          print_newline();
                          print_newline()))
```

s'évalue en :

```
                print_string "\nnote d'ecrit: ";
                        let e = read_float() in
                                (print_string "\nnote d'oral: ";
                                        let o = read_float() in

                ((fun a b -> let r= (3.*.a+.b)/. 4. in
                        print_string (if r<10. then "ajourne" else "admis");
                        print_newline();
                        print_newline()) e o))
```

L'évaluation ci-dessus provoque l'évaluation de :

```
        - print_string "\nnote d'ecrit: ";
        - read_float()
        - print_string "\nnote d'oral: ";
        - read_float()
        - ((fun a b -> let r= (3.*.a+.b)/. 4. in
                        print_string (if r<10. then "ajourne" else "admis");
                        print_newline();
                        print_newline()) e o)
```

où l'environnement contient maintenant 2 nouvelles liaisons pour `e` et `o` avec les 2 flottants lus au clavier.

L'évaluation de la dernière expression donne d'abord :

```
let r= (3.*.e+.o)/. 4. in
                 print_string (if r<10. then "ajourne" else "admis");
                 print_newline();
                 print_newline()
```

qui provoque enfin l'évaluation de `r`, les impressions à l'écran et qui renvoie la valeur `()`.

```
#let resultat = admission ponderation;;

note d'ecrit: 14.

note d'oral: 12.
admis

resultat : unit = ()
```

Il est préférable de définir une fonction `resultat` qui ne provoque aucune lecture et aucun affichage mais doit simplement être un raccourci d'écriture pour (admission pondération). Il faut donc retarder l'évaluation de cette expression en l'encapsulant artificiellement sous un `(fun -> ...)`.

```
#let resultat = fun () -> admission ponderation;;
resultat : unit -> unit = <fun>

#resultat();;

note d'ecrit: 8.1

note d'oral: 9.5

ajourne

- : unit = ()
#resultat();;

note d'ecrit: 14.5

note d'oral: 17.

admis

- : unit = ()
```

Il est désormais bien clair que l'évaluation d'une expression qui n'est pas purement fonctionnelle provoque des effets différents selon la stratégie adoptée.

4 Fichiers

Juste un aperçu, le lecteur pourra consulter la page 277 et suivantes de la documentation de Ocaml (version 3.08).

4.1 Fichiers en écriture

Leur nom logique est du type `out_channel` et ils sont ouverts en écriture comme suit.

```
#let c = open_out  "my_fic";;
c : out_channel = <abstr>
```

Ouvre en écriture un fichier appelé `my_fic` dans le répertoire courant et le nom du fichier physique est associé au nom logique `c`. Ce qui suit présente et illustre la fonction d'écriture.

```
#output_string;;
- : out_channel -> string -> unit = <fun>

#output_string c "hello";;
- : unit = ()

#output_string c "   bonjour!!!";;
- : unit = ()

#close_out c;;
- : unit = ()
```

Le fichier `my_fic` contient : `hello   bonjour!!!`.

```
#let c= open_out "my_fic";;
c : out_channel = <abstr>

#output_string c "je l'ignore
";;
- : unit = ()

#output_string c "maintenant je sais
 et je vais a la ligne";;
- : unit = ()

#output_string c "\nfausse_fin";;
```

```
- : unit = ()

#close_out c;;
- : unit = ()
```

Ici le fichier `my_fic` contient ceci :

```
je l'ignore
maintenant je sais
 et je vais a la ligne
fausse_fin
```

Le contenu précédent a été écrasé.

4.2 Fichiers en lecture

Voici le traitement d'un fichier en lecture.

```
#let c=open_in "my_fic";;
c : in_channel = <abstr>

#input_char c ;;
- : char = 'j'

#input_char c;;
- : char = 'e'

#input_line c;;
- : string = " l'ignore  "

#input_line c;;
- : string = "maintenant je sais"

#input_line c;;
- : string = " et je vais a la ligne"

#input_line c;;
- : string = "fausse_fin"

#input_line c;;
Uncaught exception: End_of_file
```

Le contenu de `my_fic` n'a pas changé.

```
#let c=open_out "my_fic";;
c : out_channel = <abstr>

#close_out c;;
- : unit = ()
```

Ici `my_fic` est vide.

5 Structures de données modifiables

5.1 Les références

Une référence est en fait un pointeur au sens de C ou Pascal. C'est donc une adresse mémoire.

– On crée une référence de la façon suivante :
```
#let p= ref 0;;
p : int ref = ref 0
```
`p` désigne donc un pointeur vers une case mémoire de type entier. Mais à la création, on doit donner une valeur initiale qui détermine aussi le type de la variable crée.

– On accède à la valeur de cette variable par `!` :
```
#!p;;
- : int = 0
```

– On modifie la valeur de la variable pointée par `:=` :
```
#p:= !p+1;;
- : unit = ()
```

On peut ainsi créer une fonction d'incrémentation :

```
#let incr ptr = ptr:= !ptr+1;;
incr : int ref -> unit = <fun>

#!p;;
- : int = 1
#incr p;;
- : unit = ()
#!p;;
- : int = 2
```

Remarquer la différence :

```
#let x=1;;
x : int = 1                          ici x est une valeur qui vaut 1

#let f y = x+y;;
f : int -> int = <fun>               par suite f est la fonction y->1+y

#let x = 2;;
x : int = 2                          ici x est une valeur qui vaut 2 et
                                     l'ancienne valeur de x est ecrasée.
#f 2;;
- : int = 3                          ... mais f n'a pas été modifiée!

#let x = ref 1;;
x : int ref = ref 1          x est un pointeur vers une case contenant 1

#let f y= !x + y;;
f : int -> int = <fun>       f est la fonction qui ajoute à y ce qu'il y
                             a dans cette case
#f 2;;
- : int = 3

#x:= 2;;
- : unit = ()                on met 2 dans x

#f 2;;
- : int = 4                 Tant qu'on ne change pas la valeur de x,
                            f est la fonction y -> 2 + y
```

Attention aux eta-réductions !

Tant que l'on était en fonctionnel, on pouvait faire des eta-réductions, c'est-a-dire remplacer `(fun x-> (f x))` par `f`. Ca n'est plus vrai quand on programme avec des effets de bords.

Exemple.

```
#let f = fun x -> incr x; (fun z -> z+1);;
f : int ref -> int -> int = <fun>
```

Cette fonction incrémente `x` puis renvoie la fonction successeur.

Définissons maintenant :

```
#let compteur= ref 0;;
compteur : int ref = ref 0

#let g y = f compteur y;;
g : int -> int = <fun>
```

Lorsque l'on définit `g` de cette façon , la valeur pointée par compteur ne change pas puisqu'il n'y a pas d'évaluation sous un `(fun y -> ...)`.

```
#!compteur;;
- : int = 0
```

À chaque appel de `g` sur des valeurs particulières, la variable pointée par compteur est incrémentée et donc cette variable contient à chaque instant le nombre d'appels à `g` déjà effectues.

```
#g 3;;
- : int = 4
#g 2;;
- : int = 3
#g 10;;
- : int = 11
#!compteur;;
- : int = 3
```

Maintenant si l'on fait ce qu'on croit être une eta-réduction, on définit

```
#let g = f compteur;;
g : int -> int = <fun>
```

L'expression `f compteur` est évaluée puisque ça n'est pas une abstraction. Par suite, compteur est incrémenté au moment de la définition de `g`, et ce, une fois pour toutes. La valeur de compteur ne changera plus.

```
#!compteur;;
- : int = 4

#g 0;;
- : int = 1
#g 1;;
- : int = 2
#g 2;;
```

```
- : int = 3

#!compteur;;
- : int = 4
```

Donc la fonction obtenue par eta-réduction n'est pas équivalente à la première.

5.2 Égalité des valeurs et égalité physique

Jusqu'à présent nous n'avons vu qu'une seule égalité en Caml, notée = et qui compare les valeurs des 2 expressions en membre droit et en membre gauche du signe =. Par exemple :

```
#let x=1.1;;
x : float = 1.1
#let y=1.1;;
y : float = 1.1
#x=y;;
- : bool = true
```

Il existe une autre égalité, qui n'a aucun intérêt d'un point de vue purement fonctionnel, qui teste si les 2 membres de l'égalité représente une même adresse mémoire. Elle est notée == et est intéressante du point de vue implémentation.

```
#x==y;;
- : bool = false
#let z=x;;
z : float = 1.1
#z==x;;
- : bool = true
```

Donc, une déclaration :

```
#let z=x;;
```

ne crée pas un nouvel objet mais est une nouvelle façon de nommer un objet qui existe déjà.

Si `p1` et `p2` sont 2 références, `p1=p2` testera l'égalité des valeurs des variables pointées par `p1` et `p2` respectivement. Quand à l'égalité `p1==p2`, elle compare les adresses portées par `p1` et `p2`.

Exemple.

```
#let p1= ref 1;;
```

```
p1 : int ref = ref 1

#let p2 = ref 1;;              |      #let p3=p1;;
p2 : int ref = ref 1           |      p3 : int ref = ref 16
#p1=p2;;                       |
- : bool = true                |
#p1==p2;;                      |      #p3==p1;;
- : bool = false               |      - : bool = true
                               |
#p1:= !p1 + 15;;               |      #p1:=!p1+3;;
- : unit = ()                  |      - : unit = ()
#p1;;                          |      #p1;;
- : int ref = ref 16           |      - : int ref = ref 19
#p2;;                          |      #p3;;
- : int ref = ref 1            |      - : int ref = ref 19
```

La construction `as`**.** Dans le même esprit, la construction `as` peut être utilisée pour représenter un sous-motif d'un motif.

Par exemple l'expression `(fun ((x,y),z) -> (x,y))` peut être remplacée par :

```
(fun ((x,y) as t, z) -> t)
```

Le résultat est alors l'argument `(x,y)`. On ne construit pas une nouvelle paire. `t` est l'adresse de l'argument et ce procède permet donc d'économiser de la place mémoire.

```
#let id = fun ((x,y) as t )->t;;
id : 'a * 'b -> 'a * 'b = <fun>

#let x = (1,1);;
x : int * int = 1, 1

#x == id x;;
- : bool = true

#let id1 = fun (x,y)->(x,y);;
id1 : 'a * 'b -> 'a * 'b = <fun>

#x == id1 x;;
- : bool = false
```

5.3 Vecteurs

Le type `vect` est prédéfini. Il est générique c'est-à-dire qu'en fait le type de vecteurs est `'a vect`.

Notations :

```
#[|0;1;2;3|];;
- : int vect = [|0; 1; 2; 3|]

#[|'a';'b';'c';'d'|];;
- : char vect = [|'a'; 'b'; 'c'; 'd'|]

#[|"vos";"yeux";"belle";"marquise"|];;
- : string vect = [|"vos"; "yeux"; "belle"; "marquise"|]

#[|1;'a'|];;
Toplevel input:
>[|1;'a'|];;
>    ^^^
This expression has type char,
but is used with type int.
```

Vecteur vide :

```
#[||];;
- : 'a vect = [||]
```

À la différence des listes :
– il n'y a pas de constructeur. Les vecteurs sont définis en une seule fois.
– il y a un accès direct à l'élément de rang `p` (le premier élément d'un vecteur a pour rang `0`).

```
#let v=[|1;2;3;4;5|];;
v : int vect = [|1; 2; 3; 4; 5|]

#v.(0);;
- : int = 1

#v.(4);;
- : int = 5

#v.(10);;
Uncaught exception: Invalid_argument "vect_item"
```

```
#[||].(0);;
Uncaught exception: Invalid_argument "vect_item"
```

– les éléments d'un vecteur sont modifiables. L'opération d'affectation est : `v.(i)<- e`. Ceci sort complètement du cadre fonctionnel.

```
#let v=[|1;2;3;4;5|];;
v : int vect = [|1; 2; 3; 4; 5|]

#v.(0)<- 10;;
- : unit = ()

#v;;
- : int vect = [|10; 2; 3; 4; 5|]
```

Donc, et contrairement aux liaisons qui restent dans le cadre purement fonctionnel, l'expression `v` et l'expression `[|1 ;2 ;3 ;4 ;5|]` introduite au moment de la définition de `v` ne sont plus interchangeables.

Voici quelques fonctions prédéfinies sur les vecteurs.

Longueur :

```
#Array.length;;
- : 'a vect -> int = <fun>

#(Array.length [||]);;
- : int = 0
#Array.length [|1;2|];;
- : int = 2
```

Fonction `Array.map` :

```
#Array.map;;
- : ('a -> 'b) -> 'a vect -> 'b vect = <fun>
#Array.map (fun x->x+1) [|2;4;6;8|];;
- : int vect = [|3; 5; 7; 9|]
```

Fonctions `Array.to_list` et `Array.of_list` :

```
#Array.to_list;;
- : 'a vect -> 'a list = <fun>

#(Array.to_list [|1;2;3;4|]);;
- : int list = [1; 2; 3; 4]
```

```
#Array.of_list;;
- : 'a list -> 'a vect = <fun>

#Array.of_list [1;2;3;4];;
- : int vect = [|1; 2; 3; 4|]
```

Nous pouvons definir une fonction `map_vect_list` :

```
#let map_vect_list f vec = Array.to_list (Array.map f vec);;
map_vect_list : ('a -> 'b) -> 'a vect -> 'b list = <fun>

#map_vect_list (fun x -> x+1) [|2;4;6;8|];;
- : int list = [3; 5; 7; 9]
```

Fonction `Array.iter` :

```
#Array.iter;;
- : ('a -> 'b) -> 'a vect -> unit = <fun>

#Array.iter print_int (Array.map (fun x->2*x) [|1;3;5;7|]);;
261014- : unit = ()

#Array.iter (fun n -> (print_int(n);print_char(' ')))[|1;2;3|];;
1 2 3 - : unit = ()
```

Exemple.

Recherche dichotomique dans un vecteur d'éléments triés.

```
#let dichotomie comp e v =
 let rec dicho d f =  if f<d then false
                      else
                        let m=(d+f)/2 in
                            if e=v.(m) then true
                            else
                                if (comp e v.(m)) then (dicho  d (m-1))
                                          else (dicho  (m+1) f)
 in dicho 0 (Array.length v -1) ;;
dichotomie : ('a -> 'a -> bool) -> 'a -> 'a vect -> bool = <fun>


# dichotomie (<=)  "yeux" [|"belle"; "d'amour"; "font"; "marquise"; "me"; "mourir";
```

```
                                "vos"; "yeux"|];;

- : bool = true

# dichotomie (<=) "mes"  [|"belle"; "d'amour"; "font"; "marquise"; "me"; "mourir";
                                "vos"; "yeux"|];;

- : bool = false
```

Échange de 2 éléments dans un vecteur.

```
#let swap v i j =
    let l= (Array.length v)-1 in
        if (i>l) or (j>l) then (failwith "swap")
        else
           let x =v.(i) in
             (v.(i)<-v.(j);
              v.(j)<-x);;

swap : 'a vect -> int -> int -> unit = <fun>

#let v =  [|1;2;3;4;5;6;7;8|];;
v : int vect = [|1; 2; 3; 4; 5; 6; 7; 8|]

#swap v 0 5;;
- : unit = ()

#v;;
- : int vect = [|6; 2; 3; 4; 5; 1; 7; 8|]
```

5.4 Enregistrements

Enregistrements simples. C'est un type produit dont les composantes peuvent avoir des types différents.

```
#type date = {mois:int;annee:int};;
Type date defined.
```

`mois` et `annee` sont les étiquettes de l'enregistrement.

```
#let m={mois=1;annee=1952};;
```

```
m : date = {mois = 1; annee = 1952}

#m.mois;;
- : int = 1

#type individu= {nom:string;arrivee:date};;
Type individu defined.

#let toto={nom="toto";arrivee={mois=1;annee=1952}};;

toto : individu = {nom = "toto"; arrivee = {mois = 1; annee = 1952}}

#toto.arrivee.annee;;
- : int = 1952
```

On peut utiliser le filtrage sur les enregistrements :

```
#let bissextile = fun {mois =_; annee=x} -> x mod 4 = 0;;
bissextile : date -> bool = <fun>

#bissextile {mois=1;annee=1952};;
- : bool = true

#bissextile {mois=1;annee=1997};;
- : bool = false
```

Enregistrements à champs modifiables. Certains champs de l'enregistrement peuvent être mutables. Ceci permet de modifier la valeur de ce champ, comme pour les vecteurs.

De même qu'il est possible de modifier des éléments d'un vecteur, il est possible de modifier la valeur des champs d'un enregistrement à condition qu'ils aient été déclarés de type mutable.

Par exemple, un point du plan peut être repéré par ses coordonnées ;

```
#type coord = {mutable abs:float; mutable ord:float};;
Type coord defined.
```

Ceci est équivalent à la définition :

```
#type coord = {abs:float ref ; ord:float ref};;
Type coord defined.
```

Les objets de ces 2 types sont strictement isomorphes. Les 2 champs contiennent des adresses. Seule la syntaxe diffère. Les déclarations avec champs mutables induisent une syntaxe plus simple que celles avec références, comme illustré par ce qui suit.

Définissons la translation de vecteur de coordonnées `(a,b)` :

```
#let translation (a,b) pt = pt.abs <- pt.abs+.a; pt.ord <- pt.ord+.b;;

translation : float * float -> coord -> unit = <fun>

#let point={abs=0. ; ord=3.};;
point : coord = {abs = 0.0; ord = 3.0}

#translation (1.,1.) point;;
- : unit = ()

#point;;
- : coord = {abs = 1.0; ord = 4.0}
```

Faisons subir un changement à `p1` et examinons le résultat sur `p2` et `p3` définis ci-après :

```
#let p1 = {abs= 2.; ord= 2.};;
p1 : coord = {abs= 2.0; ord= 2.0}

#let p2 = {abs= 2.; ord= 2.};;
p2 : coord = {abs= 2.0; ord= 2.0}

#let p3 = p1;;
p3 : coord = {abs= 2.0; ord= 2.0}

#translation (1.,1.) p1;;
- : unit = ()

#p1;;
- : coord = {abs = 3.0; ord = 3.0}

#p2;;
- : coord = {abs = 2.0; ord = 2.0}

#p3;;
- : coord = {abs = 3.0; ord = 3.0}
```

6 Exemples : les listes chaînées circulaires

Les listes chaînées sont constituées de cellules qui sont des enregistrements à 2 champs : l'un portant une information, l'autre l'adresse de la cellule suivante. Le second champ étant une

adresse, il sera donc mutable.

```
#type 'a cellule={info:'a; mutable suiv:'a cellule};;
Type cellule defined.
```

La liste étant circulaire, la dernière cellule contiendra donc l'adresse de la première.

Création d'une cellule.

```
#let rec toto = {info= 1; suiv=toto};;
toto : int cellule =
 {info = 1;
  suiv =
   {info = 1;
    suiv =
     {info = 1;
      suiv =
      ....}}}}}}}}}}}}}}}}
```

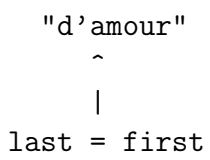
Création d'une liste circulaire

```
#let make_list e = let rec x={info=e;suiv=x} in x;;
make_list : 'a -> 'a cellule = <fun>

#let l= make_list "d'amour";;
l : string cellule =
 {info = "d'amour";
  suiv =
   {info = "d'amour";
    suiv =
     {info = "d'amour";
          ....}}}}}}}}}}}}}}}}
```

L'idée est de repérer la liste par la dernière cellule. Ainsi, on peut faire des insertions en tête en rajoutant une cellule juste après la dernière. Les déclarations ci-dessous donnent les informations portées respectivement par la dernière et la première cellule d'une liste chaînée `l`.

```
#let last l = l.info;;
last : 'a cellule -> 'a = <fun>

#let first l = l.suiv.info;;
first : 'a cellule -> 'a = <fun>
```

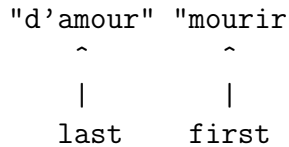
On a en fait :



Insertion en tête :

```
#let insert l e = l.suiv <- {info=e;suiv=l.suiv};l;;
insert : 'a cellule -> 'a -> 'a cellule = <fun>

#let l= insert l "mourir";;
l : string cellule =
 {info = "d'amour";
  suiv =
   {info = "mourir";
    suiv =
       ....}}}}}}}}}}}}}}}}
```

On a :




```
#last l;;
- : string = "d'amour"
#first l;;
- : string = "mourir"

#let l= insert l "font";insert l "me";;
l : string cellule =
 {info = "d'amour";
  suiv =
   {info = "me";
    suiv =
     {info = "font";
      suiv =
       {info = "mourir";
        suiv =
```

```
          {info = "d'amour";
           suiv =
            {info = "me";
                        ....}}}}}}}}}}}}}}}}
```

État de la liste :

```
                    "d'amour" "me" "font" "mourir"
                        ^      ^
                        |      |
                       last   first
```

```
#last l;;
- : string = "d'amour"

#first l;;
- : string = "me"

#let l = insert l "yeux"; insert l "beaux";insert l "vos";;
l : string cellule =
 {info = "d'amour";
  suiv =
   {info = "vos";
    suiv =
     {info = "beaux";
      suiv =
       {info = "yeux";
        suiv =
         {info = "me";
          suiv =
           {info = "font";
            suiv =
             {info = "mourir";
              suiv =  ....}}}}}}}}}}}}}}}}
```

État de la liste :

```
                  "d'amour" "vos" "beaux" "yeux" "me" "font" "mourir"
                      ^       ^
                      |       |
                     last   first
```

```
#first l;;
- : string = "vos"
#last l;;
- : string = "d'amour"
```

Insertion en queue :

```
(* Variante *)
#let insert_tail l e = l.suiv <- {info=e;suiv=l.suiv}; l.suiv;;
                       (l.suiv<- x;x);;
#let insert_tail l e = let x = {info=e;suiv=l.suiv} in
                       (l.suiv<- x;x);;
insert_tail : 'a cellule -> 'a -> 'a cellule = <fun>

#let l= insert_tail l "belle";;
l : string cellule =
 {info = "belle";
  suiv =
   {info = "vos";
    suiv =
     {info = "beaux";
      suiv =
       {info = "yeux";
        suiv =
         {info = "me";
          suiv =
           {info = "font";
            suiv =
             {info = "mourir";
              suiv =
               {info = "d'amour";
                suiv =
                 {info = "belle";

                       ....}}}}}}}}}}}}}}}}
```

On a :

```
          "belle" "vos" "beaux" "yeux" "me" "font" "mourir" "d'amour"
            ^       ^
            |       |
          last   first
```

```
#first l;;
- : string = "vos"

#last l;;
- : string = "belle"

#let l = insert_tail l "marquise";;

l : string cellule =
 {info = "marquise";
  suiv =
   {info = "vos";
    suiv =
     {info = "beaux";
      suiv =
       {info = "yeux";
        suiv =
         {info = "me";
          suiv =
           {info = "font";
            suiv =
             {info = "mourir";
              suiv =
               {info = "d'amour";
                suiv =
                 {info = "belle";
                  suiv =
                   {info = "marquise";

                         ....}}}}}}}}}}}}}}}}
#first l;;
- : string = "vos"

#last l;;
- : string = "marquise"
```

On a :

```
  "marquise" "vos" "beaux" "yeux" "me" "font" "mourir" "d'amour" "belle"
       ^       ^
       |       |
      last   first
```

Suppression de la tête :

```
#let elim_head l = l.suiv <- l.suiv.suiv; l;;
elim_head : 'a cellule -> 'a cellule = <fun>
```

Attention, ça n'élimine la tête que si `l` contient au moins 2 éléments.

```
#elim_head l;;
- : string cellule =
 {info = "marquise";
  suiv =
   {info = "beaux";
    suiv =
     {info = "yeux";
      suiv =
       {info = "me";
        suiv =
         {info = "font";
          suiv =
           {info = "mourir";
            suiv =
             {info = "d'amour";
              suiv =
                     ....}}}}}}}}}}}}}}}}

#let m = make_list 1;;
m : int cellule =
 {info = 1;
  suiv =
   {info = 1;
    suiv =
            ....}}}}}}}}}}}}}}}}

#elim_head m;;
- : int cellule =
 {info = 1;
  suiv =
   {info = 1;
    suiv =
     {info = 1;
```

Comment reconstituer la liste chaînée `l` définie précédemment à partir de la liste :

```
 ["vos";"beaux";"yeux";"me";"font";"mourir";"d'amour";"belle";"marquise"] ?
```

On crée une liste chaînée `l` de premier élément `"vos"`. Puis on insère en queue chaque élément de la liste.

Rappel :

```
(fold_left f a [b1;...;bn]) renvoie (f....(f (f a b1) b2)...bn)


#let l= let x= make_list "vos" in
fold_left insert_tail
        x
        ["beaux";"yeux";"me";"font";"mourir";"d'amour";"belle";"marquise"] ;;

l : string cellule =
 {info = "marquise";
  suiv =
   {info = "vos";
    suiv =
     {info = "beaux";
      suiv =
       {info = "yeux";
        suiv =
         {info = "me";
          suiv =
           {info = "font";
            suiv =
             {info = "mourir";
              suiv =
               {info = "d'amour";
                suiv =
                 {info = "belle";
                  suiv =
                            ....}}}}}}}}}}}}}}}

#let circ_list_of_list =   function []   -> failwith "circ_list_of_list"
                         | (a::l) ->let x= make_list a in
                                    (List.fold_left insert_tail x l);;
circ_list_of_list : 'a list -> 'a cellule = <fun>

#let q = circ_list_of_list ["vos";"beaux";"yeux";"me";"font";"mourir";"d'amour";
                            "belle";"marquise"] ;;
q : string cellule =
```

```
 {info = "marquise";
  suiv =
   {info = "vos";
    suiv =
     {info = "beaux";
      suiv =
       {info = "yeux";
        suiv =
         {info = "me";
          suiv =
           {info = "font";
            suiv =
             {info = "mourir";
              suiv =
               {info = "d'amour";
                suiv =
                 {info = "belle";
                  suiv =
                   {info = "marquise";
                    suiv =
                     ...
                        "; suiv = .}}}}}}}}}}}}}}}}
```

Réciproquement, la liste des éléments d'une liste circulaire peut être obtenue de la façon suivante :

```
#let list_of_circ_list l =
     let rec l_of_c p =
         if p==l then [] else
            first p :: l_of_c p.suiv
     in
           (first l :: l_of_c l.suiv) ;;
list_of_circ_list : 'a cellule -> 'a list = <fun>

#list_of_circ_list q;;
- : string list =
 ["vos"; "beaux"; "yeux"; "me"; "font"; "mourir"; "d'amour"; "belle";
  "marquise"]
```

7 Boucles

Pour finir, rappelons que Caml nous donne la possibilité de construire des boucles à l'aide des mots clés `for` et `while`. La syntaxe est la suivante :

```
for nom = expr1 to expr2 do expr3 done
for nom = expr1 downto expr2 do expr3 done
```

où `expr1,expr2 : int` et `expr3 : unit`. Par exemple :

```
# let liste i n =
  let ls = ref [] in
    for j = n downto i
    do
      ls := j::!ls
    done ;
    !ls;;
            val liste : int -> int -> int list = <fun>

# liste 1 10;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

La syntaxe de `while` est la suivante :

```
while expr1 do expr2 done
```

où `expr1 : bool` et `expr2 : unit`. Un exemple d'utilisation est :

```
# let until predicat changer x =
  let y = ref x in
    while not (predicat !y)
    do
      y :=  changer(!y)
    done;
    !y;;
            val until : ('a -> bool) -> ('a -> 'a) -> 'a -> 'a = <fun>

# until (fun x -> x > 10) (fun x -> x+1) 0;;
- : int = 11
```

Pour être cohérent, le corps de la boucle (`expr3` pour `for` et `expr2` pour `while`) doit avoir le type `unit`. Autrement Ocaml nous donne un avertissement :

```
# let until predicat changer x =
  let y = ref x in
    while not (predicat !y)
    do
      y :=  changer(!y) ; 1
    done;
    !y;;
            Characters 92-113:
Warning: this expression should have type unit.
        y :=  changer(!y) ; 1
        ^^^^^^^^^^^^^^^^^^^^
val until : ('a -> bool) -> ('a -> 'a) -> 'a -> 'a = <fun>
```

En effet l'expression `y := changer( !y) ; 1` a type `int`, comme on peut aisément s'en convaincre :

```
# let y = ref 0;;
val y : int ref = {contents = 0}

# y:= !y + 1; 1;;
- : int = 1
```