Structures mutables

G. Dewaele

7 janvier 2026

Lycée Louis-le-Grand

Programmation fonctionnelle

Pour l'instant, on a utilisé un style *fonctionnel*

En programmation fonctionnelle :

- on applique des fonctions à des constantes
- on utilise la récursion
- on ne contrôle généralement pas l'ordre d'exécution
- on ne gère pas la mémoire

Programmation impérative

En programmation impérative :

- on travaille avec des données variables
- les instructions altèrent leur valeur
- on utilise des boucles
- on contrôle l'ordre d'exécution
- on peut avoir à gérer la mémoire

Programmation fonctionnelle

```
let rec pgcd a = function
  | 0 -> a
  | b -> pgcd b (a mod b);;
```



```
pgcd 24 42
```

Programmation fonctionnelle

```
let rec pgcd a = function
  | 0 -> a
  | b -> pgcd b (a mod b);;
```



```
pgcd 24 42
= pgcd 42 24
```

Programmation fonctionnelle

```
let rec pgcd a = function
  | 0 -> a
  | b -> pgcd b (a mod b);;
```



```
pgcd 24 42
= pgcd 42 24
= pgcd 24 18
```

Programmation fonctionnelle

```
let rec pgcd a = function
  | 0 -> a
  | b -> pgcd b (a mod b);;
```



```
pgcd 24 42
= pgcd 42 24
= pgcd 24 18
= pgcd 18 6
```

Programmation fonctionnelle

```
let rec pgcd a = function
  | 0 -> a
  | b -> pgcd b (a mod b);;
```



```
pgcd 24 42
= pgcd 42 24
= pgcd 24 18
= pgcd 18 6
= pgcd 6 0
```

Programmation fonctionnelle

```
let rec pgcd a = function
  | 0 -> a
  | b -> pgcd b (a mod b);;
```



```
pgcd 24 42
= pgcd 42 24
= pgcd 24 18
= pgcd 18 6
= pgcd 6 0
= 6
```

Programmation impérative

```python
def pgcd(a, b) :
    while b != 0 :
        a, b = b, a % b
    return a
```

```
pgcd(24, 42)
```

```
a = 24
```

```
b = 42
```

Programmation impérative

```python
def pgcd(a, b) :
    while b != 0 :
        a, b = b, a % b
    return a
```

```
pgcd(24, 42)
```

```
a = 42
```

```
b = 24
```

Programmation impérative

```python
def pgcd(a, b) :
    while b != 0 :
        a, b = b, a % b
    return a
```

```
pgcd(24, 42)
```

```
a = 24
```

```
b = 18
```

Programmation impérative

```python
def pgcd(a, b) :
    while b != 0 :
        a, b = b, a % b
    return a
```

```
pgcd(24, 42)
```

```
a = 18
```

```
b = 6
```

Programmation impérative

```python
def pgcd(a, b) :
    while b != 0 :
        a, b = b, a % b
    return a
```

```
pgcd(24, 42)
```

```
a = 6
```

```
b = 0
```

Programmation impérative

```python
def pgcd(a, b) :
    while b != 0 :
        a, b = b, a % b
    return a
```

```
pgcd(24, 42)

a = 6

b = 0

return 6
```

Programmation multi-paradigmes

Tout algorithme peut être écrit dans les deux styles

(parfois avec une complexité différente)

En général, on mélange les styles !

On choisira ce qui est le plus simple/lisible

Sinon, chacun ses préférences...

Séquences d'instructions

Les instructions en séquences sont séparées par `;`

```
# let foo x =
    print_string "Le carré de "; print_int x;
    print_string " est "; print_int (x * x);
    print_string ".";
    print_newline ();;

val foo : int -> unit = <fun>
```

Séquences d'instructions

C'est un simple raccourci pour


```
# let foo x =
    let _ = print_string "Le carré de " in
    let _ = print_int x in
    let _ = print_string " est " in
    let _ = print_int (x * x) in
    let _ = print_string "." in
    print_newline ();;

val foo : int -> unit = <fun>
```

Séquences d'instructions

La résultat de la séquence est le résultat de la dernière instruction


```
# let foo x =
    print_string "Calcul du carré de ";
    print_int x; print_newline ();
    x * x;;

val foo : int -> int = <fun>
```

Séquences d'instructions

La résultat de la séquence est le résultat de la dernière instruction

```
# let foo x =
    print_string "Calcul du carré de ";
    print_int x; print_newline ();
    x * x;;

val foo : int -> int = <fun>
```

```
# foo 2;;
Calcul du carré de 2
- : int = 4
```

Séquences d'instructions

On peut mettre des séquences n'importe où :


```
# min (print_string "Hello "; 42)
      (print_string "World!"; let x=29 in x+8);;

World!Hello - : int = 37
```

Séquences d'instructions

On peut mettre des séquences n'importe où :

```
# min (print_string "Hello "; 42)
      (print_string "World!"; let x=29 in x+8);;

World!Hello - : int = 37
```


La première séquence « vaut » 42

Séquences d'instructions

On peut mettre des séquences n'importe où :

```
# min (print_string "Hello "; 42)
      (print_string "World!"; let x=29 in x+8);;

World!Hello - : int = 37
```


La première séquence « vaut » 42

La seconde séquence « vaut » 37

Séquences d'instructions

On peut mettre des séquences n'importe où :

```
# min (print_string "Hello "; 42)
      (print_string "World!"; let x=29 in x+8);;

World!Hello - : int = 37
```


La première séquence « vaut » 42

La seconde séquence « vaut » 37

Les instructions des séquences sont exécutées dans l'ordre

Mais on ne sait quelle séquence est évaluée en premier !

Séquences d'instructions

Seule la valeur de la dernière instruction est utile

Toutes les autres devraient retourner `()`

```
# let foo x =
    x * x * x;
    x * x;;

Characters 18-27:
    x * x * x;
    ^^^^^^^^^
Warning 10: this expression should have type unit.
val foo : int -> int = <fun>
```

Séquences d'instructions

Attntion aux erreurs !

```
# let foo x =
    x = x + 1;   (* ce n'est PAS une incrémentation *)
    x * x;;



Characters 18-27:
     x = x + 1;
     ^^^^^^^^^
Warning 10: this expression should have type unit.
val foo : int -> int = <fun>
```

Séquences d'instructions

Il faudrait écrire, pour obtenir $(x+1)^2$:


```
# let foo x =
    let x = x + 1 in
      x * x;;

val foo : int -> int = <fun>
```

Conditions

Rappelons que l'on dispose d'un **if ... then ... else ...**

```
if condition then expression_1 else expression_2
```



```
# let rec fact n =
    if n <= 1
      then 1
      else n * fact (n-1);;

val fact : int -> int = <fun>
```

Conditions

En programmation impérative, cela sert à contrôler l'exécution :


```
# let foo n =
    print_int n;
    print_string " est ";

    if n mod 2 = 1 then
      print_string "impair"
    else
      print_string "pair";

    print_newline();;

val foo : int -> unit = <fun>
```

Conditions

Une seule expression doit suivre **then** et **else** :


```
# let foo n =
    print_int n;
    print_string " est ";

    if n mod 2 = 1 then
      print_string "impair"; print_newline ()
    else
      print_string "pair"; print_newline ();;

Characters 95-99:
    else
    ^^^^
Error: Syntax error
```

Blocs d'instructions

Il faut donc ici construire un « bloc » d'instructions :


```
# let foo n =
    print_int n;
    print_string " est ";

    if n mod 2 = 1 then
      (print_string "impair"; print_newline ())
    else
      (print_string "pair"; print_newline ());;

val foo : int -> unit = <fun>
```

Blocs d'instructions

Les parenthèses peuvent être remplacées par **begin ... end**


```
# let foo n =
    print_int n;
    print_string " est ";

    if n mod 2 = 1 then
      begin
        print_string "impair";
        print_newline ()
      end
    else
      begin
        print_string "pair";
        print_newline ()
      end;;

val foo : int -> unit = <fun>
```

Blocs d'instructions

begin ... **end** et (...) sont pratiquement interchangeables

```
# 2 * begin 3 + 4 end;;  (* A proscrire ! *)
- : int = 14
```


On respectera l'usage :

- des parenthèses s'il s'agit de priorités dans une expression
- **begin** ... **end** s'il s'agit d'un bloc d'exécution

La « valeur » associée à un bloc est celle de la dernière instruction

Blocs d'instructions et filtrage

Dans le cas suivant, où m1, m2, m3 et m4 sont des motifs :

```
match expr1 with
  | m1 -> match expr2 with
            | m2 -> ...
            | m3 -> ...
  | m4 -> ...
```


m4 est un motif pour **match** expr2 !

En effet, l'indentation ne joue aucun rôle...

Blocs d'instructions et filtrage

On utilisera donc un bloc :


```
match expr1 with
  | m1 -> begin
            match expr2 with
              | m2 -> ...
              | m3 -> ...
          end
  | m4 -> ...
```

Cette fois, m4 est un motif pour **match** expr1

Boucles inconditionnelles

Une boucle inconditionnelle est obtenue avec un **for** :

```
for nom = expression_1 to expression_2 do
  sequence d'instructions
done
```


`expression_1` et `expression_2` doivent donner des entiers

Boucles inconditionnelles

Une boucle inconditionnelle est obtenue avec un **for** :

```
for nom = expression_1 to expression_2 do
  sequence d'instructions
done
```


`expression_1` et `expression_2` doivent donner des entiers

La boucle est effectuée de `expression_1` à `expression_2` **inclus** !

Si `expression_1 > expression_2`, l'ensemble est ignoré

Boucles inconditionnelles

Une boucle inconditionnelle est obtenue avec un **for** :

```
for nom = expression_1 to expression_2 do
  sequence d'instructions
done
```


`expression_1` et `expression_2` doivent donner des entiers

La boucle est effectuée de `expression_1` à `expression_2` **inclus** !

Si `expression_1 > expression_2`, l'ensemble est ignoré

Pas besoin de **begin** ... **end** avec le **do** ... **done**

Boucles inconditionnelles

La boucle définie par

```
for i = 1 to 4 do sequence done
```


correspond très exactement à

```
let i = 1 in sequence;
let i = 2 in sequence;
let i = 3 in sequence;
let i = 4 in sequence;
```


`i` n'est pas une variable et ne peut être changé !

Boucles inconditionnelles

Un exemple affichant une table de multiplication :


```
# let table n =
    for i = 1 to 10 do
      print_int n;
      print_string " fois ";
      print_int i;
      print_string " égale ";
      print_int (n*i);
      print_newline ();
    done;;

val table : int -> unit = <fun>
```

Boucles inconditionnelles

La séquence dans **do** ... **done** devrait retourner ()

```
# let foo n =
    for i = 1 to 10 do
      i * n
    done;;

Characters 30-35:
      i * n
      ^^^^^
Warning 10: this expression should have type unit.
val foo : int -> unit = <fun>
```

Globalement, un **for** ... **done** a toujours pour valeur ()

Boucles inconditionnelles

Il n'existe aucune manière de choisir un pas

Seule exception, les itérations décroissantes :

```
for nom = expression_1 downto expression_2 do
  sequence d'instructions
done
```

Boucles conditionnelles

Une boucle conditionnelle est obtenue avec un **while** :

```
while expression_booleenne do
  sequence d'instructions
done
```

Boucles conditionnelles

Une boucle conditionnelle est obtenue avec un **while** :

```
while expression_booleenne do
  sequence d'instructions
done
```


`expression_booleenne` doit être évaluée à ... `true` ou `false`

Si c'est `true`, la séquence est exécutée intégralement,

puis on évalue à nouveau `expression_booleenne`

Boucles conditionnelles

Un exemple :

```
while read_line () <> "Au revoir" do
  print_string "Dites m'en plus !";
  print_newline ();
done;;
```


Cette boucle affiche le même message en réponse à toute entrée

Jusqu'à ce que l'utilisateur entre « `Au revoir` »

Boucles conditionnelles

Et là... on tombe sur un os !

Boucles conditionnelles

Et là... on tombe sur un os !

Pour l'instant, on ne manipule que *des constantes* !

Entrées mises à part, `expression_booleenne` ne peut pas changer

Boucles conditionnelles

Et là... on tombe sur un os !

Pour l'instant, on ne manipule que *des constantes* !

Entrées mises à part, `expression_booleenne` ne peut pas changer

Une définition locale `let ... in ...` dans la boucle :

est autorisée, mais ne survit pas à l'itération suivante

Boucles conditionnelles

Et là... on tombe sur un os !

Pour l'instant, on ne manipule que *des constantes* !

Entrées mises à part, `expression_booleenne` ne peut pas changer

Une définition locale `let ... in ...` dans la boucle :

est autorisée, mais ne survit pas à l'itération suivante

Une définition `let ...` dans la boucle :

est interdite, comme elle l'était dans une fonction

Boucles conditionnelles

`while` n'a de sens qu'en programmation *impérative*

On a absolument besoin de pouvoir modifier l'état de la mémoire

Il nous faut des « *variables* »

De l'usage du point-virgule

`;` sert à séparer des expressions/instructions

Il est inutile juste avant un **`end`**, un **`done`**, un **`else`**

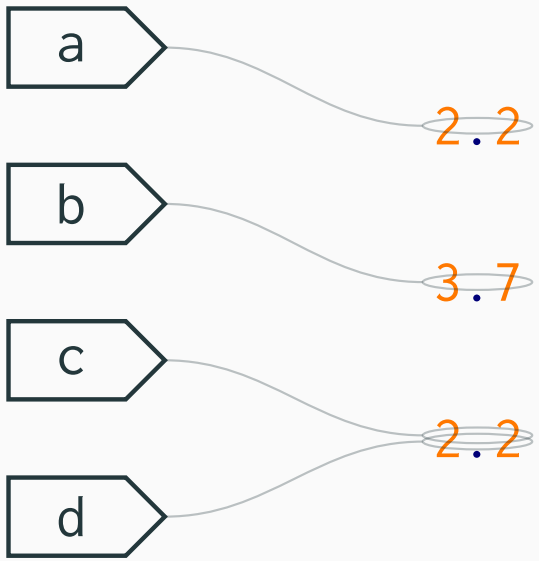
On peut en avoir besoin *après* un **`end`** ou un **`done`**

En effet, **`begin ... end`**, **`for ... done`** et **`while ... done`** se comportent, globalement, comme une (grosse) expression

Les références

Considérons :

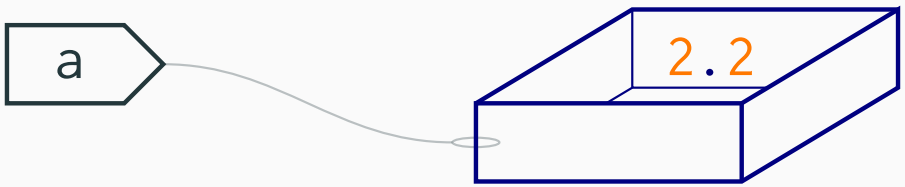
```
# let a = 2.2 and b = 3.7 and c = 2.2;;
# let d = c;;
```



Les références

Pour permettre le changement, on associe le nom à une « boîte » :

```
# let a = ref 2.2;;
val a : float ref = {contents = 2.2}
```



On parle en OCaml de *référence*

L'association entre le nom et la boîte est définitif

Mais son *contenu* peut changer !

Les références

Un nom associé à une référence ne peut servir directement :


```
# a;;
- : float ref = {contents = 2.2}

# a *. 2.0;;

Characters 2-3:
  a *. 2.0;;
  ^
Error: This expression has type float ref
       but an expression was expected of type float
```

Les références

Pour obtenir l'élément dans la boite a, on utilise !


```
# a;;
- : float ref = {contents = 2.2}

# !a;;
- : float = 2.2

# !a *. 2.0;;
- : float = 4.4
```

Les références

Pour placer un élément dans la boîte, on utilise `:=`

```
# a;;
- : float ref = {contents = 2.2}

# a := 3.7;;
- : unit = ()

# a;;
- : float ref = {contents = 3.7}
```

S'il n'y avait pas de nom associé à l'élément initialement dans la boîte, celui-ci est perdu !

Les références

Les « boîtes » sont associées à un type précis (ex. : `float ref`)

Celui-ci ne peut pas changer

Une « boîte » n'est *jamais vide*

Les références

On peut avoir des références avec tous les types

Par exemple des fonctions :


```
# let funct = ref abs;;
val funct : (int -> int) ref = {contents = <fun>}

# !funct (-37);;
- : int = 37

# funct := fun x -> x*x*x;;
- : unit = ()

# funct := min 0;;
- : unit = ()
```

Les références

Des références de listes :

```
# let r = ref [ 1; 2 ];;
val r : int list ref = {contents = [1; 2]}
```

Le type des objets dans la liste ne pourra pas changer !

Si on crée la référence avec une liste vide :

```
# let r = ref [];;
val r : '_weak1 list ref = {contents = []}
```

Le type sera fixé lorsque introduira la première liste non-vide

Les références

Inversement, on peut créer des listes de références :

```
# let lst = [ ref 1; ref 2 ];;
val lst : int ref list = [{contents = 1};
                          {contents = 2}]
```


Ou bien même une référence vers une liste de références :

```
# let r = ref [ ref 1; ref 2 ];;
val r : int ref list ref = {contents =
                              [{contents = 1};
                               {contents = 2}]}
```

Les références

On peut même créer des références de références :


```
# let a = ref (ref 42);;
val a : int ref ref = {contents = {contents = 42}}

# !a;;
- : int ref = {contents = 42}

# a := ref 22
- : unit = ()

# ! !a;;              (* OCaml ne comprendrait pas !!a *)
- : int = 42

# !a := 37;;
- : unit = ()
```

Exemples

Pour calculer une factorielle dans un style impératif :


```
# let fact n =
    let accum = ref 1 in
      for i = 2 to n do
        accum := !accum * i
      done;
      !accum;;

val fact : int -> int = <fun>
```

Exemples

Pour calculer un pgcd dans un style impératif :


```
# let pgcd u v =
    let a = ref u and b = ref v in
      while !b <> 0 do
        let tmp = !a in          (*                        *)
          a := !b;               (* a, b <- b, a mod b *)
          b := tmp mod !b        (*                        *)
      done;
      !a;;

val pgcd : int -> int -> int = <fun>
```

Incrémentation, décrémentation

`incr` et `decr` sont des fonctions de signature `int ref -> unit`

`incr i` est équivalent à `i := !i + 1`

`decr i` est équivalent à `i := !i - 1`


```
# let i = ref 17;;
val i : int ref = {contents = 17}

# incr i;;
- : unit = ()

# i;;
- : int ref = {contents = 18}
```

Exemples

Pour compter les zéros dans une liste d'entiers :

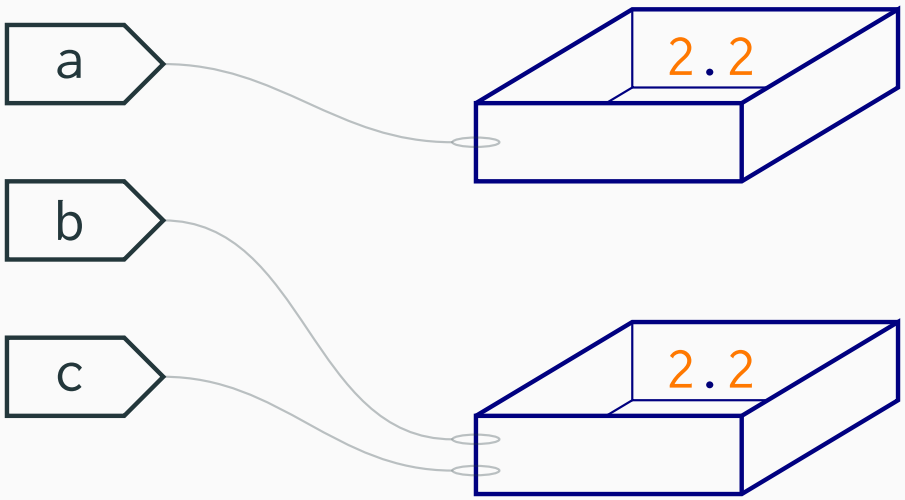

```
# let compte_zeros lst =
    let nombre = ref 0 and reste = ref lst in
      while !reste <> [] do
        if List.hd !reste = 0 then incr nombre;
        reste := List.tl !reste
      done;
      !nombre;;

val compte_zeros : int list -> int = <fun>
```

Égalité, identité

Considérons :

```
# let a = ref 2.2 and b = ref 2.2;;
# let c = b;;
```



Égalité, identité

Considérons :

```
# let a = ref 2.2 and b = ref 2.2;;
# let c = b;;
```


Les opérateurs = et <> testent *l'égalité des contenus*

```
# a = b;;
- : bool = true    (* car !a = !b *)

# a <> b;;
- : bool = false
```

Égalité, identité

Considérons :

```
# let a = ref 2.2 and b = ref 2.2;;
# let c = b;;
```


Les opérateurs == et != testent *l'identité des références*

```
# a == b;;
- : bool = false   (* références distinctes ! *)

# b == c;;
- : bool = true    (* deux noms, même référence *)
```

Égalité, identité

Si l'on veut être précis, l'image de la boîte a des limites

Un *même* objet peut être simultanément dans plusieurs boîtes

Dans l'exemple suivant :

```
# let a = ref 2.2 and b = ref 2.2
# let c = ref !b
```

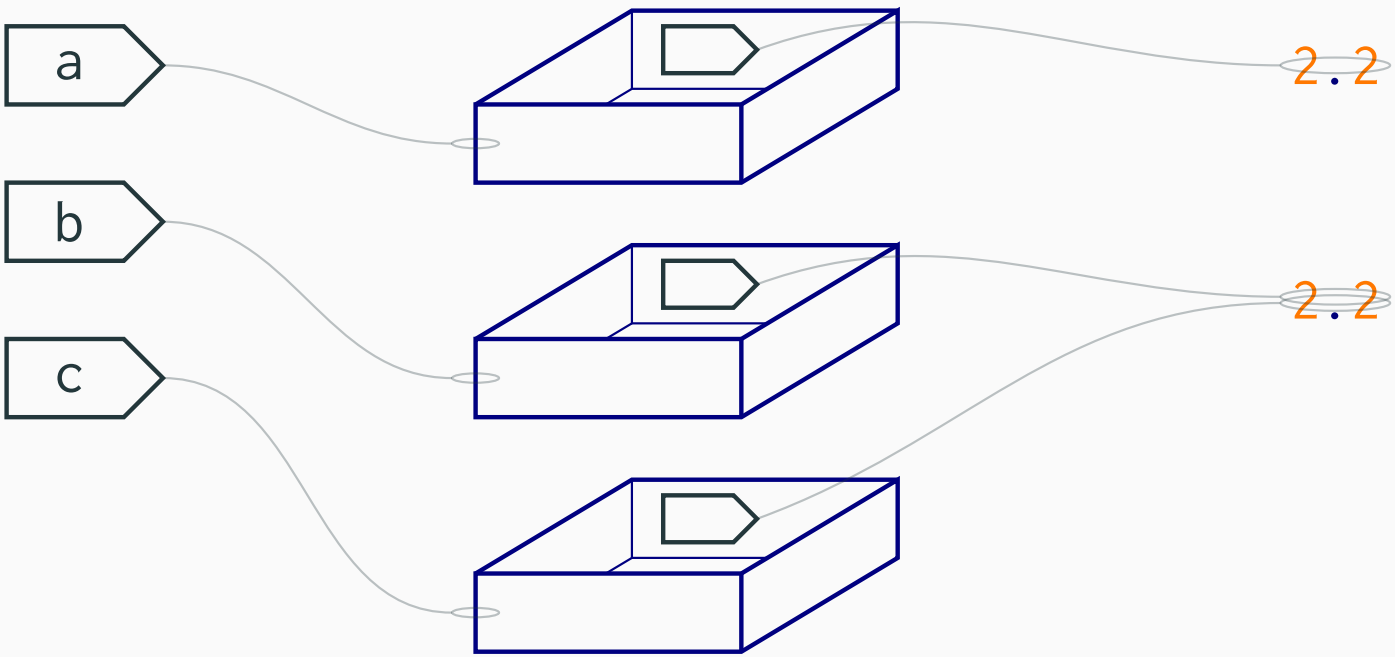

On a trois boîtes distinctes

Mais celles désignées par b et c contiennent le *même* 2.2

Égalité, identité

Dans l'exemple suivant :

```
# let a = ref 2.2 and b = ref 2.2
# let c = ref !b
```



Structures mutables

Les références sont des objets *mutables*

Il existe d'autre objets mutables en OCaml

Ils visent à simplifier la programmation impérative

Un cas concret

On souhaite réaliser un annuaire

Un cas concret

On souhaite réaliser un annuaire

nom	numéro
Dupont	1234
Durand	2211
Martin	6789

Un cas concret

On souhaite réaliser un annuaire

nom	numéro
Dupont	1234
Durand	2211
Martin	6789

```
type coord = { name: string ; number: string }
```

Solution fonctionnelle

On peut représenter l'annuaire comme un coord `list` :


```
let phonebook = [
  { name = "Dupont" ; number = "0123456789" } ;
  { name = "Durand" ; number = "0246813579" } ;
  { name = "Martin" ; number = "0918273645" }
]
```

Nous étudierons plus tard une représentation plus efficace

Solution fonctionnelle

Pour modifier un numéro :


```
# let rec update name new_number = function
    | h::t when h.name = name
        -> { name = name ; number = new_number }
              :: update name new_number t
    | h::t -> h :: update name new_number t
    | [] -> [];;

val update : string -> string -> coord list
                                -> coord list = <fun>
```

Solution fonctionnelle

On construit un *nouvel annuaire*


```
# let new_phonebook =
    update "Durand" "0000012345" phonebook;;

val new phonebook : coord list =
  [{name = "Dupont"; number = "0123456789"};
   {name = "Durand"; number = "0000012345"};
   {name = "Martin"; number = "0918273645"}]
```

Limites

Problème : certains noms peuvent désigner l'ancien annuaire

Ils sont associés à des données qui ne sont plus à jour !

On peut vouloir *modifier* l'annuaire existant

Une solution à base de références

On peut utiliser des références :


```
type coord = { name: string ; number: string ref }

let phonebook = [
  { name = "Dupont" ; number = ref "0123456789" } ;
  { name = "Durand" ; number = ref "0246813579" } ;
  { name = "Martin" ; number = ref "0918273645" }
]
```

Une solution à base de références

La modification de l'annuaire devient :


```
# let rec update name new_number = function
    | h::t when h.name = name
        -> h.number := new_number;
           update name new_number t
    | h::t -> update name new_number t
    | [] -> ();;

val update : string -> string -> coord list
                                -> unit = <fun>
```

Une solution à base de références

On remarque que le résultat est cette fois un **unit** :

```
# modifie "Durand" "0000056789" annuaire;;
- : unit = ()

# annuaire;;
- : coord list =
[{nom = "Dupont"; numéro = {contents = "0123456789"}};
 {nom = "Durand"; numéro = {contents = "0000056789"}};
 {nom = "Martin"; numéro = {contents = "0918273645"}}]
```

Une solution à base de références

Dans un style plus impératif :


```
# let update name new_number phonebook =
    let rest = ref phonebook in
      while !rest <> [] do
        let coord = List.hd !rest in
          if coord.name = name
            then coord.number := new_number;
          rest := List.tl !rest
      done;;

val update : string -> string -> coord list
                                -> unit = <fun>
```

Une solution à base de références

On remarque que le résultat est cette fois un **unit** :


```
...
val modifie : string -> string -> coord list
                  -> unit = <fun>

# modifie "Durand" "4237" annuaire;;
- : unit = ()

# annuaire;;
- : coord list =
[{nom = "Dupont"; numero = {contents = "1234"}};
{nom = "Durand"; numero = {contents = "4237"}};
{nom = "Martin"; numero = {contents = "6789"}}]
```

Une solution à base de références

Avec cette approche, on substitue aux chaînes des références

Il faut donc ajouter des ! partout où le numéro est utilisé

Une solution à base de références

Avec cette approche, on substitue aux chaînes des références

Il faut donc ajouter des ! partout où le numéro est utilisé

C'est potentiellement assez lourd

Une solution à base de références

Avec cette approche, on substitue aux chaînes des références

Il faut donc ajouter des ! partout où le numéro est utilisé

C'est potentiellement assez lourd

C'est du travail si on ajoute la fonctionnalité tardivement

Une autre solution

Il existe une autre solution, déclarer le champ *mutable* :

```
type coord = { name: string ; mutable number: string }

let phonebook = [
  { name = "Dupont" ; number = "0123456789" } ;
  { name = "Durand" ; number = "0246813579" } ;
  { name = "Martin" ; number = "0918273645" }
];;
```

La définition de l'annuaire est inchangée !

Une autre solution

L'utilisation est la même qu'auparavant :


```
# let c = List.hd phonebook;;
c : coord = {name = "Dupont"; number = "0123456789"}

# c.number;;
- : string = "0123456789"
```

Une autre solution

On peut *muter* le champ number avec `<-`

```
# (List.hd phonebook).number <- "9876543210";;
- : unit = ()

# List.hd phonebook;;
- : coord = {name = "Dupont"; number = "9876543210"}
```

Une autre solution

C'est le *même objet*, qui a muté !

La modification apparaît notamment dans phonebook :

```
# List.hd annuaire;;
- : coord list =
[{nom = "Dupont"; numero = "9876543210"};
{nom = "Durand"; numero = "0246813579"};
{nom = "Martin"; numero = "0918273645"}]
```

Une autre solution

Pour modifier notre annuaire, on écrit donc :


```
# let rec update name new_number = function
    | h::t when h.name = name
        -> t.number <- new_number;
           update name new_number t
    | h::t -> update name new_number t
    | [] -> ();;

val update : string -> string -> coord list
                                -> unit = <fun>
```

Une remarque pour clore

Pourquoi avoir à la fois `:=` et `<-` ?

Dans le cas

```
type foo = { mutable elem = int ref };;
```


Si x est de type `foo`, `x.elem <- ...` et `x.elem := ...` existent !

Retour sur les références

Pour l'œil averti, quand on définit une référence :

```
# let mango = ref 42;;
val mango : int ref = {contents = 42}
```


Cela ressemble à un enregistrement

Ce n'est *pas* un hasard !

Retour sur les références

Les références ne sont que du sucre syntaxique !


```
# type 'a ref = { mutable contents: 'a };;

# let ref x = { contents = x };;
val ref : 'a -> 'a ref = <fun>

# let (!) = function { contents=x } -> x;;
val ( ! ) : 'a ref -> 'a = <fun>

# let (:=) r v = r.contents <- v;;
val ( := ) : 'a ref -> 'a -> unit = <fun>
```

Les tableaux

Il est malcommode de travailler avec des `'a list` en impératif

On préfère travailler avec des `'a array`

Il s'agit de « tableaux »

Les tableaux

Les tableaux

- sont des objets mutables
- contiennent des éléments de même type
- permettent l'accès direct à un élément (en $O(1)$)
- ont une taille *fixe*

Les tableaux

Pour créer explicitement un tableau, on utilise `[| ... |]` :

```
# let arr = [| 11; 22; 37; 42; 54 |];;
val arr : int array = [|11; 22; 37; 42; 54|]
```


`Array.length` permet d'obtenir la longueur :

```
# Array.length arr;;
- : int = 5
```

Les tableaux

On accède à un élément donné de la façon suivante :

```
# arr.(3);;
- : int = 42
```

On le fait muter avec <- :

```
# arr.(3) <- 17;;
- : unit = ()

# arr;;
- : int array = [|11; 22; 37; 17; 54|]
```

Les tableaux

On peut également créer un tableau avec

- sa taille
- l'élément à mettre dans toutes les cases

Cela est réalisé avec la commande `Array.make` :

```
# Array.make 5 0.0;;
- : float array = [|0.; 0.; 0.; 0.; 0.|]
```


Comme avec « `[ elem ] * n` » en Python,

on a le *même élément* dans toutes les cases !

Les tableaux à deux dimensions

Pour créer un tableau à deux dimensions, on crée un `'a array array` :

```
# let matr = [| [| 11; 22; 37 |];
                [| 17; 42; 54 |] |];;

val matr : int array array = [|[|11; 22; 37|];
                               [|17; 42; 54|]|]
```

On accède à un élément de la sorte :

```
# matr.(0).(2);;
- : int = 37
```

Les tableaux à deux dimensions

Attention, on ne peut créer un tableau 2×3 de la sorte :

```
# let matr = Array.make 2 (Array.make 3 0.0)
val matr : float array array = [|[|0.; 0.; 0.|];
                                 [|0.; 0.; 0.|]|]
```


En effet, on a deux fois la *même ligne* !

```
# matr.(0) == matr.(1);;    (* Test d'identité ! *)
- : bool = true
```

Les tableaux à deux dimensions

Les problèmes sont évidents :

```
# matr.(0).(1) <- 42.0;;
- : unit = ()

# matr;;
- : float array array = [|[|0.; 42.; 0.|];
                          [|0.; 42.; 0.|]|]
```

Les tableaux à deux dimensions

Une solution peut être :


```
# let matr = Array.make 2 [| |];;
val matr : '_a array array = [|[||]; [||]|]

# for i=0 to 1 do matr.(i) <- Array.make 3 0.0 done;;
- : unit = ()

# matr;;
- : float array array = [|[|0.; 0.; 0.|];
                          [|0.; 0.; 0.|]|]

# matr.(0) == matr.(1);;
- : bool = false
```

Les tableaux à deux dimensions

On dispose de solutions plus simple :

```
# let matr = Array.make_matrix 2 3 0.0;;
val matr : float array array = [|[|0.; 0.; 0.|];
                                 [|0.; 0.; 0.|]|]
```

```
# let matr = Array.init 2 (fun i -> Array.make 3 0.0);;
val matr : float array array = [|[|0.; 0.; 0.|];
                                 [|0.; 0.; 0.|]|]
```

Les tableaux à deux dimensions

Il existe de nombreuses autres fonctions sur les **array**

- `Array.copy`,
- `Array.sub`,
- `Array.iter`,
- `Array.map`,
- `Array.mem`,
- `Array.to_list`,
- `Array.of_list`,
- `Array.sort`...