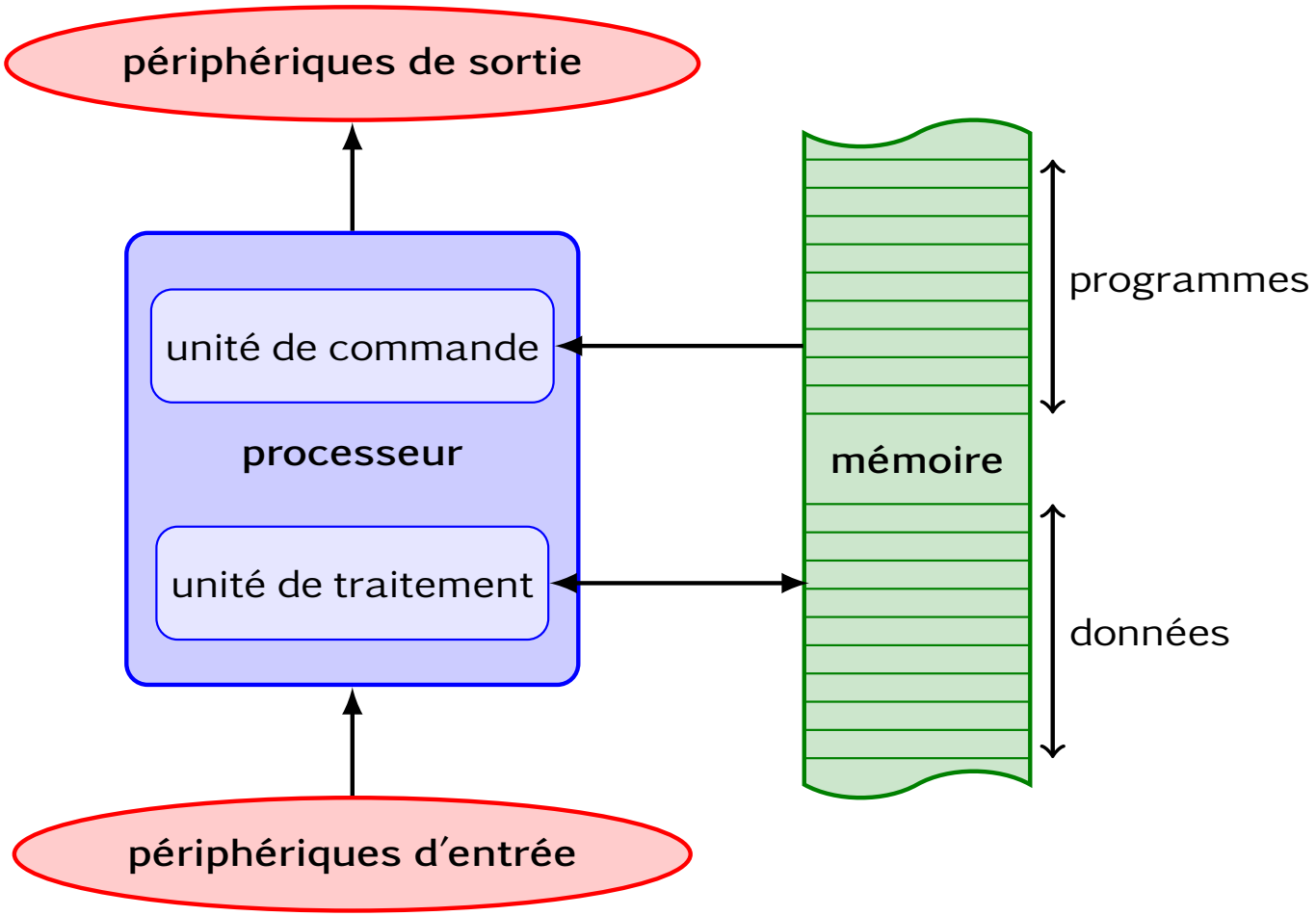
Programmation impérative

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

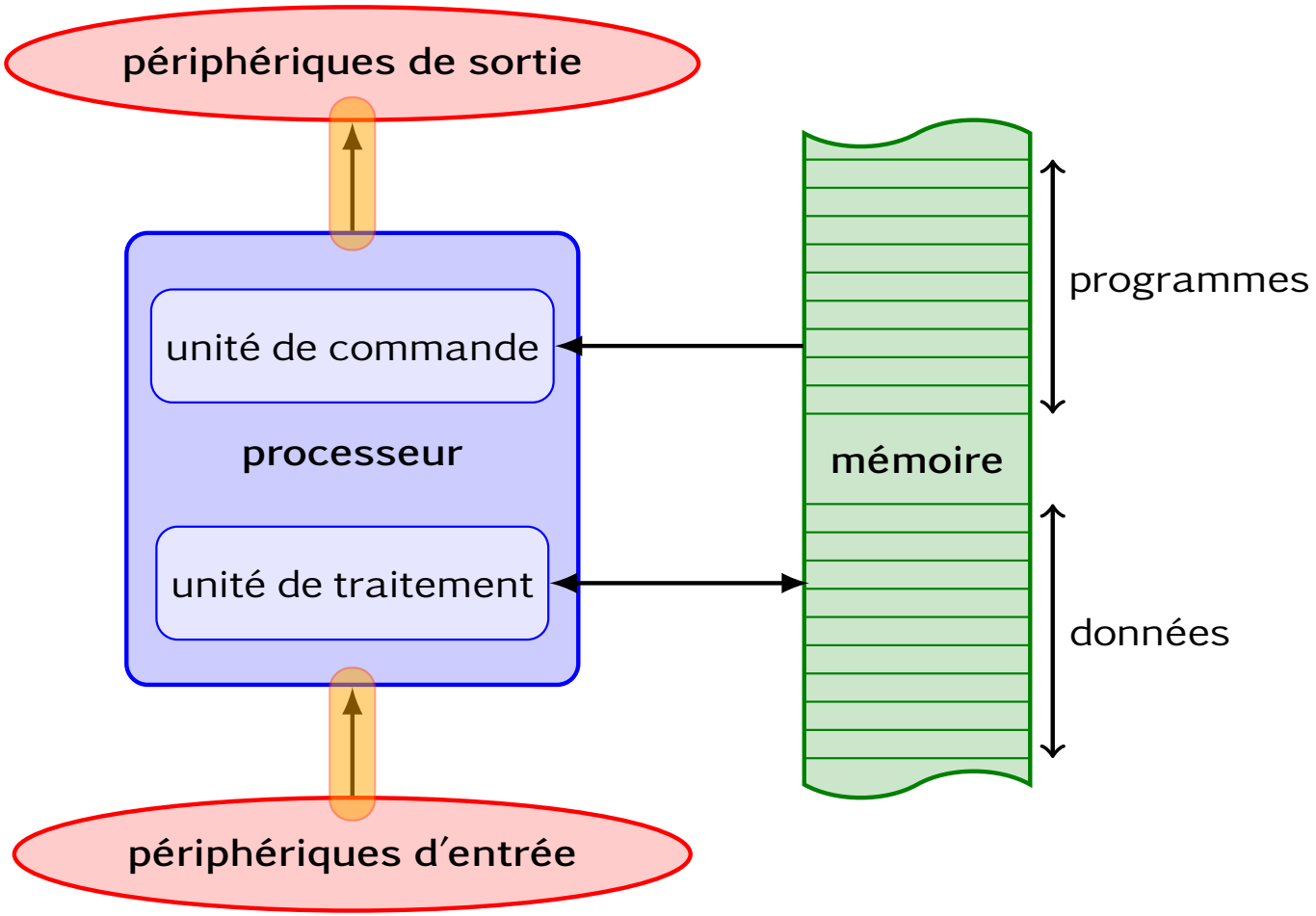
Programmation par effets

La programmation *impérative* conçoit un programme comme une séquence d'instructions ayant pour *effet* de modifier l'état de l'ordinateur.



Programmation par effets

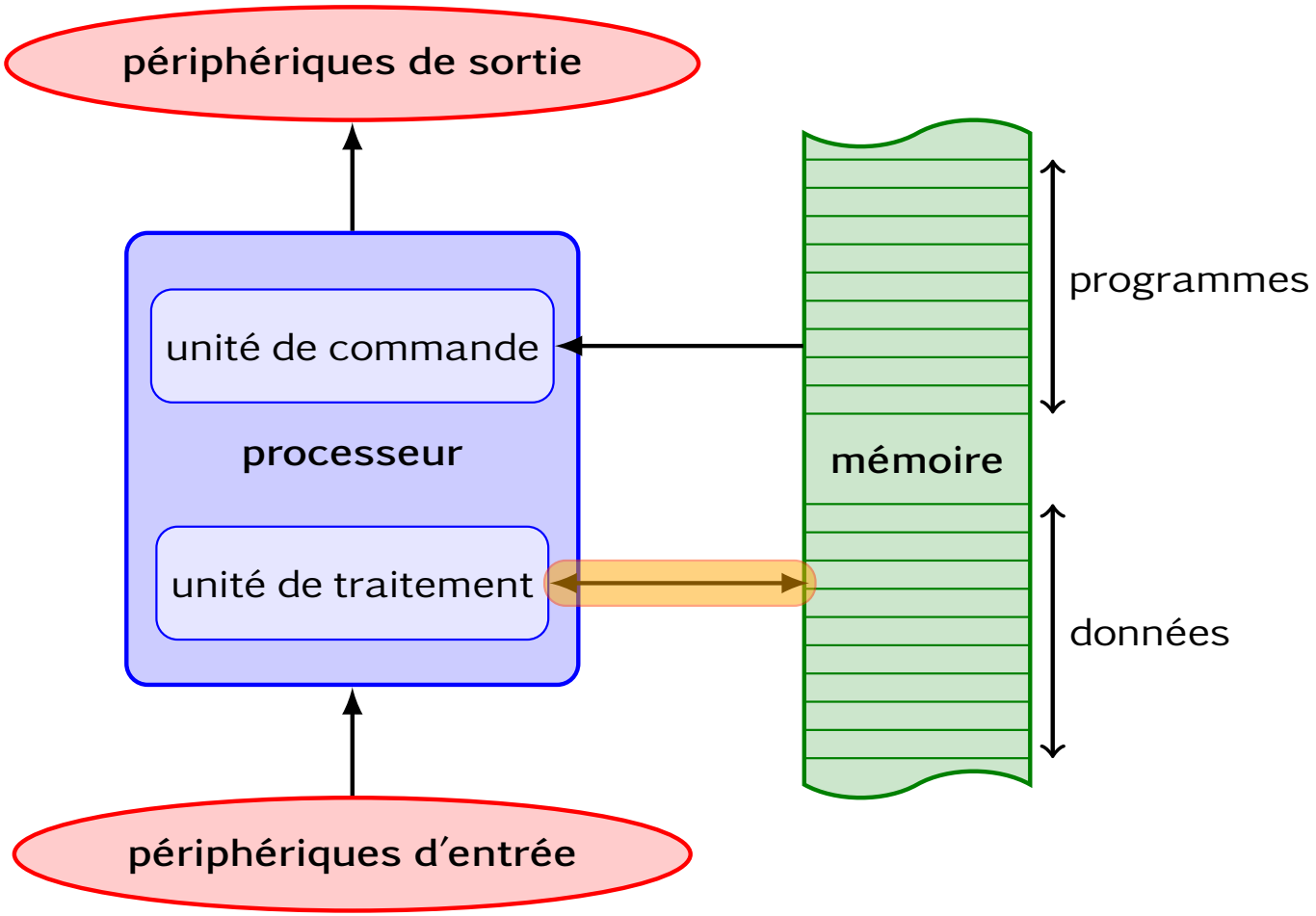
La programmation *impérative* conçoit un programme comme une séquence d'instructions ayant pour *effet* de modifier l'état de l'ordinateur.



Les effets de bords sont des interactions avec l'extérieur.

Programmation par effets

La programmation *impérative* conçoit un programme comme une séquence d'instructions ayant pour *effet* de modifier l'état de l'ordinateur.



Les effets, à proprement parler, modifient l'état de la mémoire.

Séquence d'instruction

En CAML, les séquences d'instructions sont des expressions séparées par des points-virgules.

expr1 ; expr2 ; ... ; exprn

Les effets de ces n expressions s'appliquent dans l'ordre, le résultat de la séquence est le résultat de la dernière expression.

```
# print_string "nous sommes en " ; print_int (32*9*7) ;;
nous sommes en 2016- : unit = ()
```

Séquence d'instruction

En CAML, les séquences d'instructions sont des expressions séparées par des points-virgules.

`expr1 ; expr2 ; ... ; exprn`

Les effets de ces *n* expressions s'appliquent dans l'ordre, le résultat de la séquence est le résultat de la dernière expression.

```
# print_string "nous sommes en " ; print_int (32*9*7) ;;
nous sommes en 2016- : unit = ()
```

Si on parenthèse une expression, on peut l'inclure dans un calcul :

```
# (print_int 1 ; 1) * (print_int 2 ; 2) + (print_int 3 ; 3) ;;
321- : int = 5
```

mais dans ce cas, l'ordre des effets n'est pas garanti par le langage (il suit l'ordre d'évaluation du calcul par le compilateur). Il vaut donc mieux éviter de mélanger effets et calculs.

Effets de bords

Les effets de bords sont les interactions avec les périphériques d'entrée et de sortie.

- interaction avec le terminal en sortie : **`print_int`**, **`print_float`**, **`print_char`**, **`print_string`** et **`print_newline`** ont un type de la forme *type* `->` *unit* ;
- interaction avec le terminal en entrée : **`read_int`**, **`read_float`** et **`read_line`** ont un type de la forme *unit* `->` *type* ;

mais aussi :

- interaction de sortie vers une fenêtre graphique ;
- interaction d'entrée avec une souris.

Références

Créer une référence, c'est établir un lien entre un nom et un emplacement en mémoire :

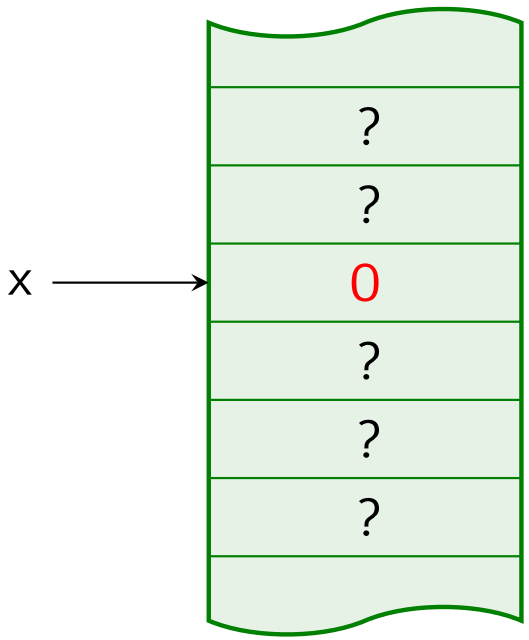
```
let identifiant = ref valeur
```

Références

Créer une référence, c'est établir un lien entre un nom et un emplacement en mémoire :

`let identifiant = ref valeur`

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

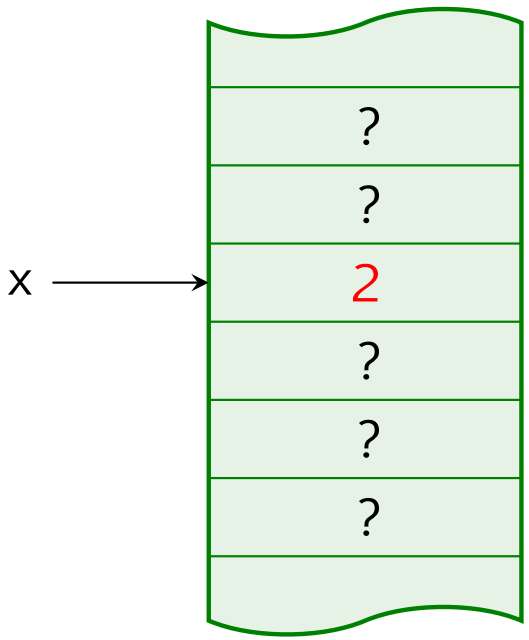


Références

Créer une référence, c'est établir un lien entre un nom et un emplacement en mémoire :

`let identifiant = ref valeur`

```
# let x = ref (0) ;;
x : int ref = ref 0
# x := 2 ;;
- : unit = ()
```

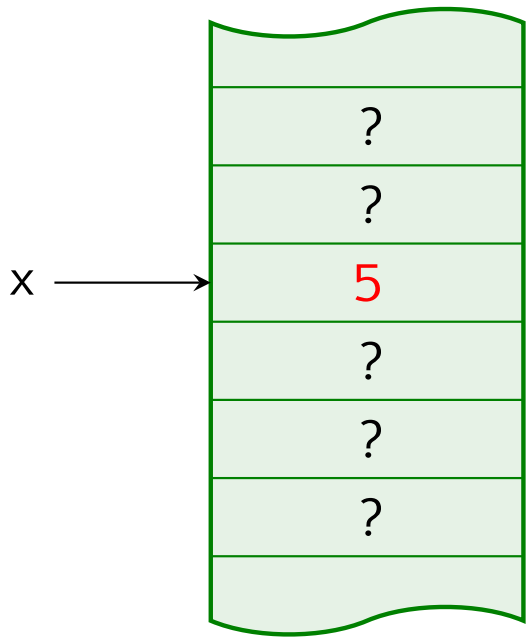


Références

Créer une référence, c'est établir un lien entre un nom et un emplacement en mémoire :

`let identifiant = ref valeur`

```
# let x = ref (0) ;;
x : int ref = ref 0
# x := 2 ;;
- : unit = ()
# x := !x + 3 ;;
- : unit = ()
```

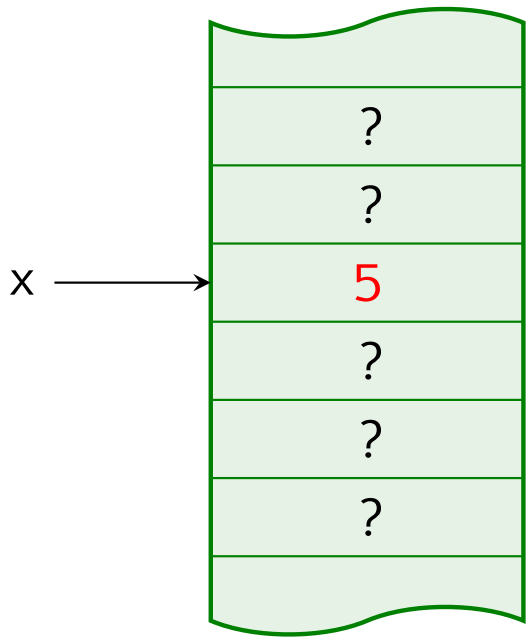


Références

Créer une référence, c'est établir un lien entre un nom et un emplacement en mémoire :

`let identifiant = ref valeur`

```
# let x = ref (0) ;;
x : int ref = ref 0
# x := 2 ;;
- : unit = ()
# x := !x + 3 ;;
- : unit = ()
# !x ;;
- : int = 5
# x ;;
- : int ref = ref 5
```



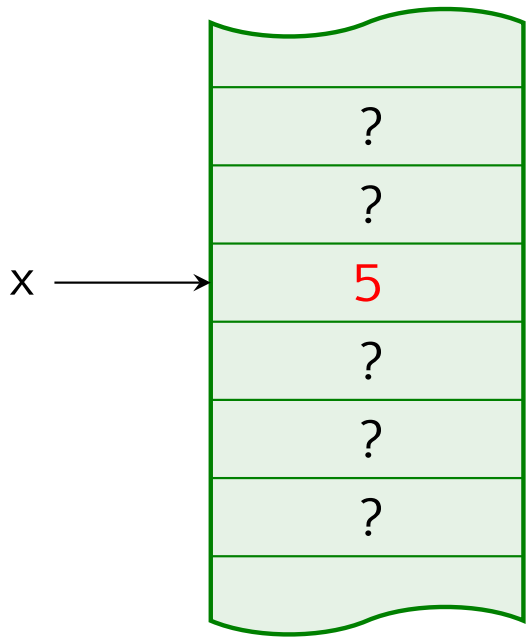
Le type d'une référence est *'a ref*, où *'a* est le type du contenu.

Références

Créer une référence, c'est établir un lien entre un nom et un emplacement en mémoire :

`let identifiant = ref valeur`

```
# let x = ref (0) ;;
x : int ref = ref 0
# x := 2 ;;
- : unit = ()
# x := !x + 3 ;;
- : unit = ()
# !x ;;
- : int = 5
# x ;;
- : int ref = ref 5
```



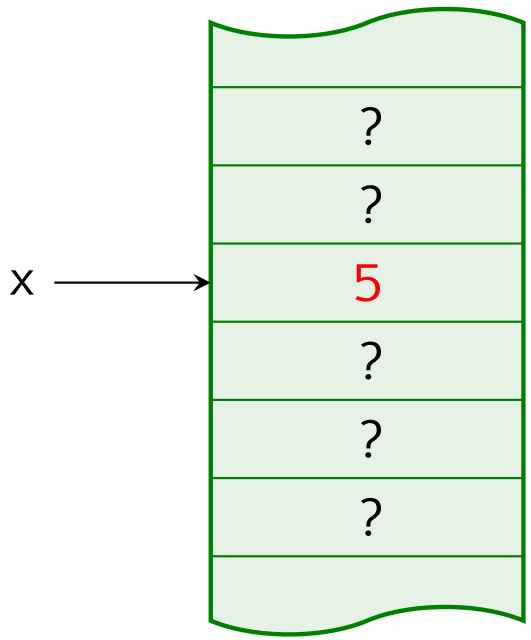

```
# prefix := ;;
- : 'a ref -> 'a -> unit = <fun>
```

Références

Créer une référence, c'est établir un lien entre un nom et un emplacement en mémoire :

`let identifiant = ref valeur`

```
# let x = ref (0) ;;
x : int ref = ref 0
# x := 2 ;;
- : unit = ()
# x := !x + 3 ;;
- : unit = ()
# !x ;;
- : int = 5
# x ;;
- : int ref = ref 5
```




```
# prefix := ;;
- : 'a ref -> 'a -> unit = <fun>
# prefix ! ;;
- : 'a ref -> 'a = <fun>
```

Différence entre définition et référence

Une définition établit une liaison entre un nom et une valeur.
Une référence établit une liaison entre un nom et un emplacement.

Différence entre définition et référence

Une définition établit une liaison entre un nom et une valeur.
Une référence établit une liaison entre un nom et un emplacement.

Une définition a un comportement *statique*, une référence un comportement *dynamique* :

```
# let a = 1 ;;
a : int = 1
# let incr x = x + a ;;
incr : int -> int = <fun>
# let a = 2 ;;
a : int = 2
# incr 0 ;;
- : int = 1
```

```
# let a = ref 1 ;;
a : int ref = ref 1
# let incr x = x + !a ;;
incr : int -> int = <fun>
# a := 2 ;;
- : unit = ()
# incr 0 ;;
- : int = 2
```

Égalité physique et structurelle

L'égalité structurelle (=) teste l'égalité entre deux valeurs.
L'égalité physique (==) détermine si les objets occupent les mêmes emplacements.

```
# let x = ref 0 ;;
x : int ref = ref 0
# let y = ref 0 ;;
y : int ref = ref 0
# !x = !y ;;                (* s'évalue en 0 = 0 *)
- : bool = true
# !x == !y ;;
- : bool = true
# x = y ;;                  (* s'évalue en ref 0 = ref 0 *)
- : bool = true
# x == y ;;
- : bool = false
```

!x et **!y** sont structurellement et physiquement égaux.
x et **y** sont structurellement égaux mais physiquement différents.

Égalité physique et structurelle

L'égalité structurelle (=) teste l'égalité entre deux valeurs.
L'égalité physique (==) détermine si les objets occupent les mêmes emplacements.

L'égalité physique permet d'illustrer de nouveau le comportement dynamique d'une référence :

```
# let x = ref 0 ;;
x : int ref = ref 0
# let z = x ;;
z : int ref = ref 0
# z == x ;;                    (* z est physiquement égal à x *)
- : bool = true
# x := 1 ;;
- : unit = ()
# !z ;;
- : int = 1
```

Types mutables

Dans un enregistrement, un champ mutable peut être modifié physiquement. Par exemple, pour définir l'équivalent d'une référence :

```
# type 'a myref = {mutable Valeur: 'a} ;;
Le type myref est défini.
```

L'équivalent des opérateurs de déréférencement `!` et d'affectation `:=` se réalisent de la façon suivante :

```
# function x -> x.Valeur ;;
- : 'a myref -> 'a = <fun>
# fun x a -> x.Valeur <- a ;;
- : 'a myref -> 'a -> unit = <fun>
```

Types mutables

Dans un enregistrement, un champ mutable peut être modifié physiquement. Par exemple, pour définir l'équivalent d'une référence :

```
# type 'a myref = {mutable Valeur: 'a} ;;
Le type myref est défini.
```

L'équivalent des opérateurs de déréférencement `!` et d'affectation `:=` se réalisent de la façon suivante :

```
# function x -> x.Valeur ;;
- : 'a myref -> 'a = <fun>
# fun x a -> x.Valeur <- a ;;
- : 'a myref -> 'a -> unit = <fun>
```

Exemple :

```
# let x = {Valeur = 0} ;;          (* équivalent de : let x = ref 0 *)
x : int myref = {Valeur = 0}
# x.Valeur <- x.Valeur + 1 ;;      (* équivalent de : x := !x + 1 *)
- : unit = ()
# x.Valeur ;;                      (* équivalent de : !x *)
- : int = 1
```

Instruction conditionnelle

Une instruction conditionnelle est construite en suivant le schéma :

`if condition then expr1 else expr2`

À l'instar de toute phrase en CAML, cette construction calcule une valeur.

```
# let x = if true then 1 else 2 ;;
x : int = 1
# if true then 1 else 2. ;;
Entrée interactive :
>if true then 1 else 2. ;;
>                       ^^
Cette expression est de type float ,
mais est utilisée avec le type int .
```

`expr1` et **`expr2`** doivent être de même type.

Instruction conditionnelle

Une instruction conditionnelle est construite en suivant le schéma :

`if condition then expr1 else expr2`

À l'instar de toute phrase en CAML, cette construction calcule une valeur. Pour cette raison, on ne peut avoir de **`if ... then ...`** sans **`else`**. Seule exception, lorsque la première expression est de type *unit* :

```
# if true then 1 ;;
Entrée interactive:
>if true then 1 ;;
>             ^
Cette expression est de type int,
mais est utilisée avec le type unit.
# if true then print_int 1 ;;
1- : unit = ()
```

Instructions d'itération

- Boucles inconditionnelles :

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

Par exemple :

```
# for i = 10 downto 1 do print_int i ; print_char '␣' done ;;
10 9 8 7 6 5 4 3 2 1 - : unit = ()
```

Instructions d'itération

- Boucles inconditionnelles :

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

Par exemple :

```
# for i = 10 downto 1 do print_int i ; print_char '␣' done ;;
10 9 8 7 6 5 4 3 2 1 - : unit = ()
```

- Boucles conditionnelles :

```
while expr1 do expr2 done
```

Par exemple :

```
# let x = ref 10 in
   while !x > 0 do
     print_int !x ; print_char '␣'; x := !x - 1 done ;;
10 9 8 7 6 5 4 3 2 1 - : unit = ()
```

Tableaux

Définition d'un tableau

- en dressant la liste de ses éléments :

```
# let t = [|'c'; 'a'; 'm'; 'l'|] ;;
t : char vect = [|'c'; 'a'; 'm'; 'l'|]
```

- à l'aide de la fonction **make_vect** :

```
# let u = make_vect 5 1 ;;
u : int vect = [|1; 1; 1; 1; 1|]
```

Tableaux

Définition d'un tableau

- en dressant la liste de ses éléments :

```
# let t = [|'c'; 'a'; 'm'; 'l'|] ;;
t : char vect = [|'c'; 'a'; 'm'; 'l'|]
```

- à l'aide de la fonction **make_vect** :

```
# let u = make_vect 5 1 ;;
u : int vect = [|1; 1; 1; 1; 1|]
```

dans un tableau de taille n, les indices s'échelonnent entre 0 et $n-1$.

Par exemple :

```
# u.(1) <- 2 ;;
- : unit = ()
# u.(2) <- u.(1) + 1 ;;
- : unit = ()
# u ;;
- : int vect = [|1; 2; 3; 1; 1|]
```

Tableaux

Définition d'un tableau

- en dressant la liste de ses éléments :

```
# let t = [|'c'; 'a'; 'm'; 'l'|] ;;
t : char vect = [|'c'; 'a'; 'm'; 'l'|]
```

- à l'aide de la fonction **make_vect** :

```
# let u = make_vect 5 1 ;;
u : int vect = [|1; 1; 1; 1; 1|]
```

dans un tableau de taille n, les indices s'échelonnent entre 0 et $n-1$.

La fonction **vect_length** retourne la taille d'un tableau :

```
# vect_length u ;;
- : int = 5
```

Parcours d'un tableau

Pour un parcours complet, on utilise une boucle inconditionnelle.
Exemple : calcul du maximum.

```
# let max_vect t =
    let n = vect_length t in
      if n = 0 then raise Not_found
      else let m = ref t.(0) in
        for k = 1 to n-1 do m := max !m t.(k) done ;
    !m ;;
max_vect : 'a vect -> 'a = <fun>
```

Parcours d'un tableau

Pour un parcours complet, on utilise une boucle inconditionnelle.
Exemple : calcul du maximum.

```
# let max_vect t =
    let n = vect_length t in
      if n = 0 then raise Not_found
      else let m = ref t.(0) in
        for k = 1 to n-1 do m := max !m t.(k) done ;
    !m ;;
max_vect : 'a vect -> 'a = <fun>
```

Pour un parcours incomplet, on utilise une boucle conditionnelle.
Exemple : recherche d'un élément.

```
# let cherche prop t =
    let rep = ref false and k = ref 0 in
      while !k < vect_length t && not !rep do rep := prop t.(!k) ;
                                              k := !k + 1  done ;
    !rep ;;
cherche : ('a -> bool) -> 'a vect -> bool = <fun>
```

Parcours d'un tableau

Pour un parcours complet, on utilise une boucle inconditionnelle.
Exemple : calcul du maximum (version fonctionnelle).

```
# let max_vect t =
      let rec aux = function
        | i when i < 0 -> raise Not_found
        | 0            -> t.(0)
        | i            -> max t.(i) (aux (i-1))
      in aux (vect_length t - 1) ;;
max_vect : 'a vect -> 'a = <fun>
```

Pour un parcours incomplet, on utilise une boucle conditionnelle.
Exemple : recherche d'un élément (version fonctionnelle).

```
# let cherche prop t =
      let rec aux = function
        | i when i < 0 -> false
        | i            -> prop t.(i) || aux (i-1)
      in aux (vect_length t - 1) ;;
cherche : ('a -> bool) -> 'a vect -> bool = <fun>
```

Comparaison entre listes et tableaux

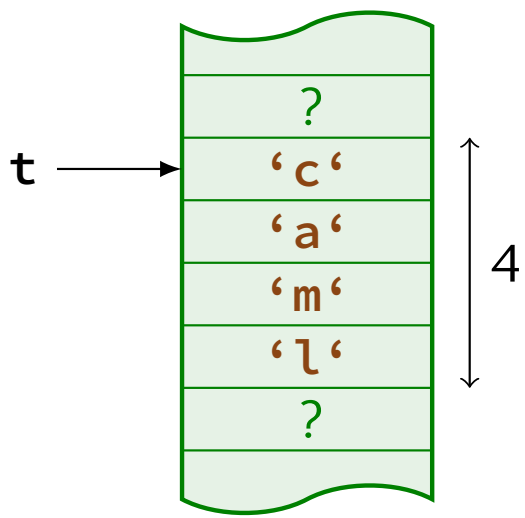
Pour une liste, modifier sa longueur a un coût constant, accéder à un élément nécessite de parcourir tous les éléments qui le précèdent.
Pour un tableau, accéder à un élément a un coût constant, modifier la longueur d'un tableau a un coût proportionnel à sa taille.

Comparaison entre listes et tableaux

Pour une liste, modifier sa longueur a un coût constant, accéder à un élément nécessite de parcourir tous les éléments qui le précèdent.
Pour un tableau, accéder à un élément a un coût constant, modifier la longueur d'un tableau a un coût proportionnel à sa taille.

Lorsqu'on crée un tableau, un bloc de n emplacements consécutifs en mémoire lui est alloué :

```
# let t = [|'c'; 'a'; 'm'; 'l'|] ;;
t : char vect = [|'c'; 'a'; 'm'; 'l'|]
```



Tableaux bi-dimensionnels

Un problème d'égalité structurelle

Une première tentative :

```
# let m = make_vect 3 (make_vect 2 0) ;;
m : int vect vect = [|[|0; 0|]; [|0; 0|]; [|0; 0|]|]
# m.(0).(0) <- 1 ;;
- : unit = ()
# m ;;
- : int vect vect = [|[|1; 0|]; [|1; 0|]; [|1; 0|]|]
```

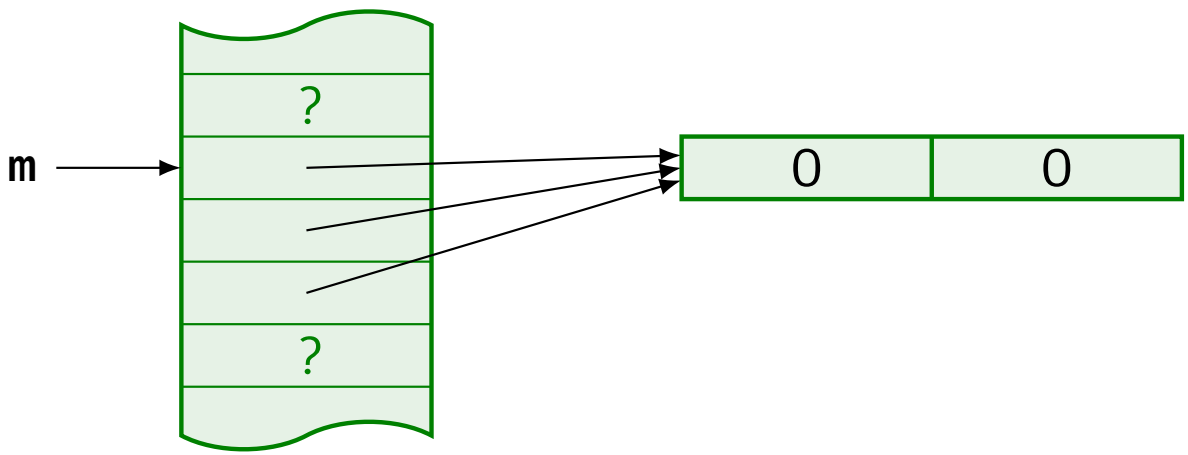
Tableaux bi-dimensionnels

Un problème d'égalité structurelle

Une première tentative :

```
# let m = make_vect 3 (make_vect 2 0) ;;
m : int vect vect = [|[|0; 0|]; [|0; 0|]; [|0; 0|]|]
# m.(0).(0) <- 1 ;;
- : unit = ()
# m ;;
- : int vect vect = [|[|1; 0|]; [|1; 0|]; [|1; 0|]|]
```

La construction effectuée :



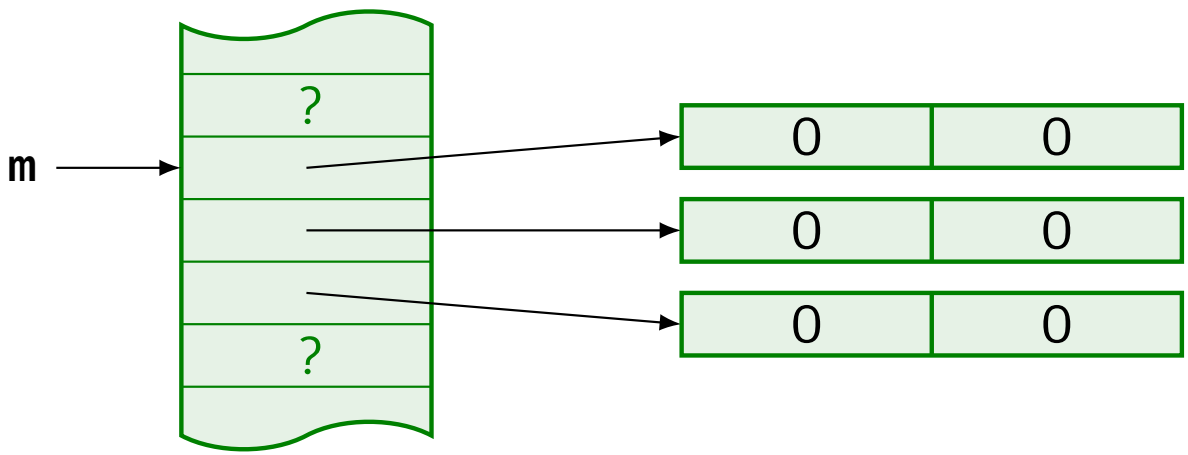
Tableaux bi-dimensionnels

Un problème d'égalité structurelle

La solution :

```
# let m = make_vect 3 [||] ;;
 m : '_a vect vect = [|[||]; [||]; [||]|]
# for i=0 to 2 do m.(i) <- make_vect 2 0 done ;;
- : unit = ()
# m ;;
- : int vect vect = [|[|0; 0|]; [|0; 0|]; [|0; 0|]|]
```

La construction effectuée :



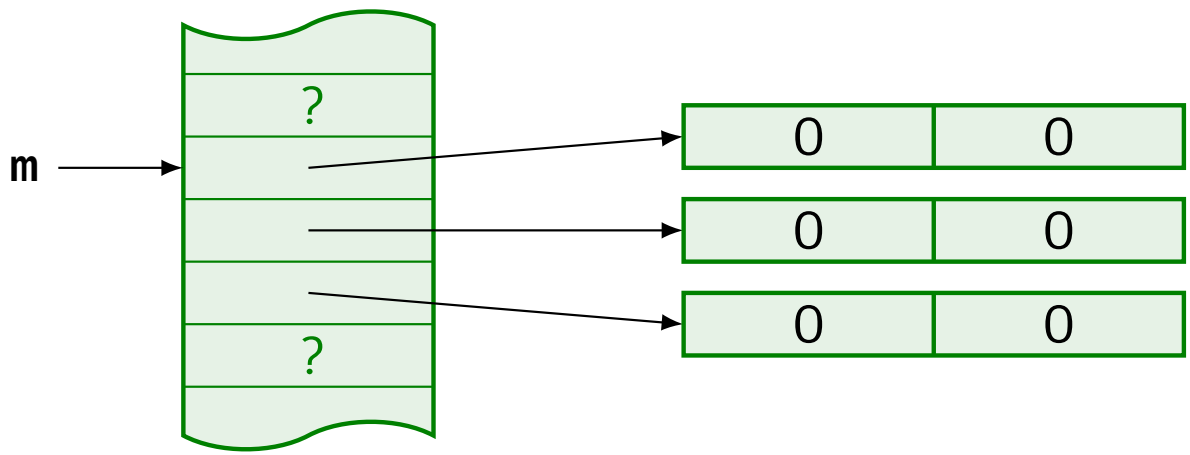
Tableaux bi-dimensionnels

Un problème d'égalité structurelle

La solution :

```
# let m = make_vect 3 [||] ;;
 m : '_a vect vect = [|[||]; [||]; [||]|]
# for i=0 to 2 do m.(i) <- make_vect 2 0 done ;;
- : unit = ()
# m ;;
- : int vect vect = [|[|0; 0|]; [|0; 0|]; [|0; 0|]|]
```

La construction effectuée :



De manière plus succincte : **`let m = make_matrix 3 2 0 ;;`**

Chaînes de caractères

Les chaînes de caractères sont elles aussi des données mutables, à l'instar des tableaux :

```
# let s1 = "abc" and s2 = "abc" in s1 == s2 ;;
- : bool = false
```

Chaînes de caractères

Les chaînes de caractères sont elles aussi des données mutables, à l'instar des tableaux :

```
# let s1 = "abc" and s2 = "abc" in s1 == s2 ;;
- : bool = false
```

Seuls différent les noms des instructions :

- **string_length** au lieu de **vect_length** ;
- **make_string** au lieu de **make_vect** ;
- **s.[i]** au lieu de **s.(i)**.

```
# let t = make_string 5 'x' ;;
t : string = "xxxxx"
# let s = "azerty" in
    s.[0] <- 'q' ;
    s.[1] <- 'w' ;
    s ;;
- : string = "qwerty"
```