Listes chaînées, Tableaux & Co.

Lycée Henri Poincaré - Nancy

10 février 2021

Sommaire

Sommaire

Mémoire

- Mémoire d'un ordinateur = suite de cases mémoires

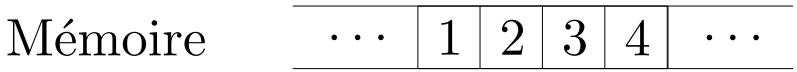
Mémoire

- Mémoire d'un ordinateur = suite de cases mémoires
- **But :** stocker les entiers `1` `2` `3` `4`

Mémoire

- Mémoire d'un ordinateur = suite de cases mémoires
- **But :** stocker les entiers `1` `2` `3` `4`

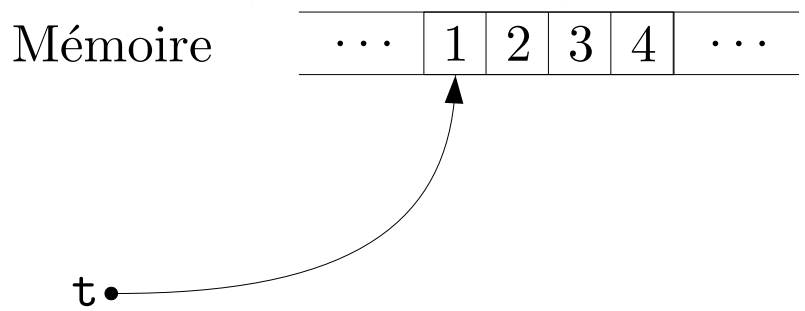
Les Tableaux



Mémoire

- Mémoire d'un ordinateur = suite de cases mémoires
- **But :** stocker les entiers `1 2 3 4`

Les Tableaux

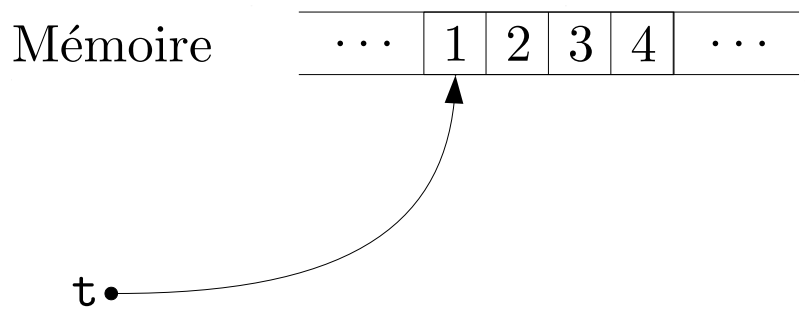



```
1  let t = [|1;2;3;4|];;
```

Mémoire

- Mémoire d'un ordinateur = suite de cases mémoires
- **But :** stocker les entiers `1 2 3 4`

Les Tableaux



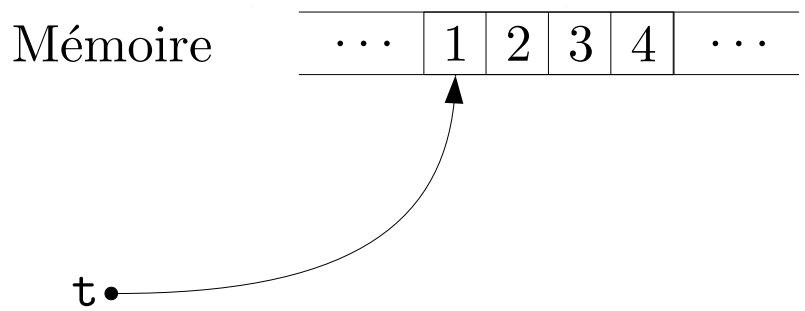

```
1  let t = [|1;2;3;4|];;
```

- Cases mémoires côte à côte

Mémoire

- Mémoire d'un ordinateur = suite de cases mémoires
- **But :** stocker les entiers `1 2 3 4`

Les Tableaux



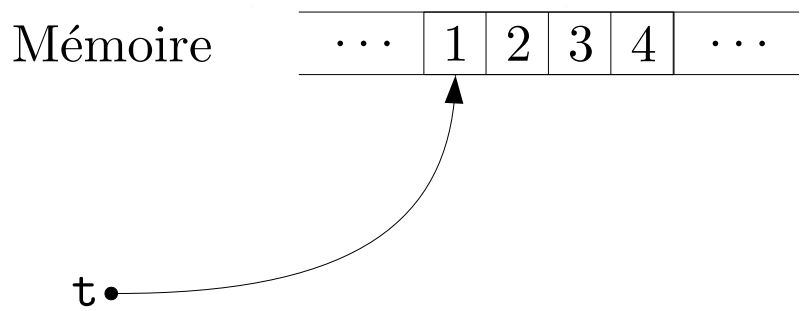

```
1  let t = [|1;2;3;4|];;
```

- Cases mémoires côte à côte
- `t` = référence vers 1ère case mémoire

Mémoire

- Mémoire d'un ordinateur = suite de cases mémoires
- **But :** stocker les entiers `1 2 3 4`

Les Tableaux



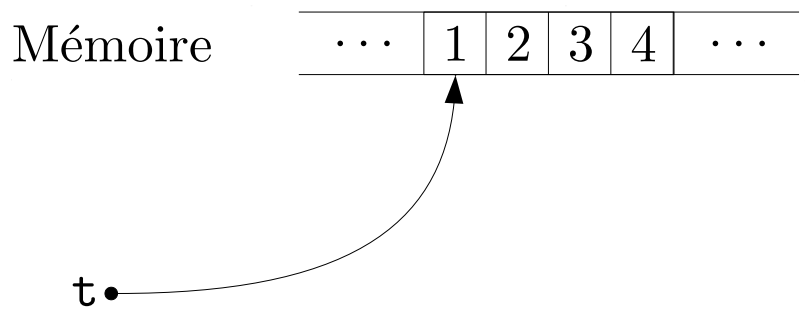

```
1  let t = [|1;2;3;4|];;
```

- Cases mémoires côte à côte
- `t` = référence vers 1ère case mémoire
- Accès au $i^{\text{ème}}$ élément : rapide

Mémoire

- Mémoire d'un ordinateur = suite de cases mémoires
- **But :** stocker les entiers `1 2 3 4`

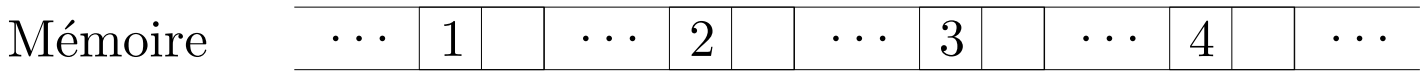
Les Tableaux




```
1  let t = [|1;2;3;4|];;
```

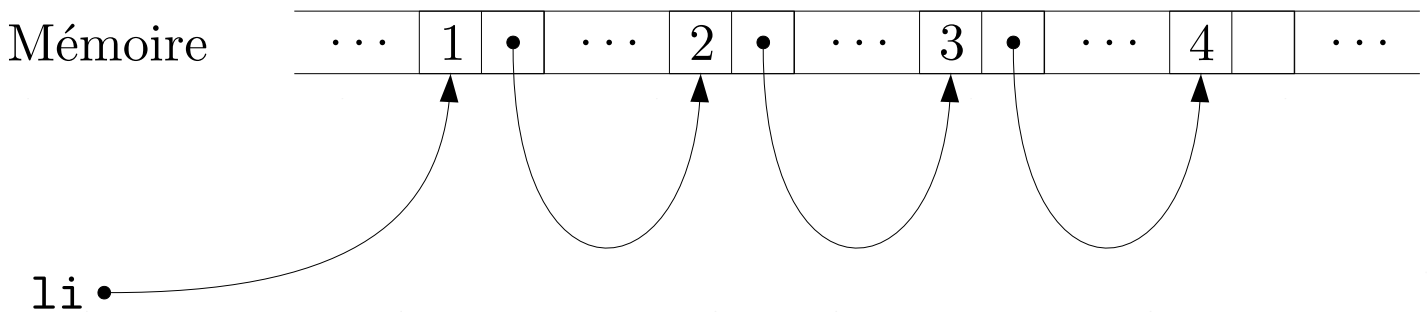
- Cases mémoires côte à côte
- `t` = référence vers 1ère case mémoire
- Accès au $i^{\text{ème}}$ élément : rapide
- Ajouter/supprimer un élément : lent

Les Listes chaînées



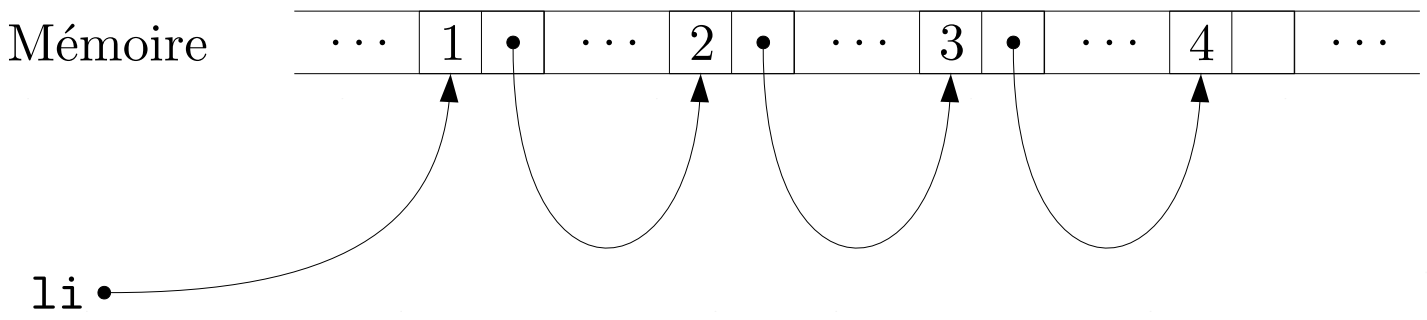
Mémoire

Les Listes chaînées



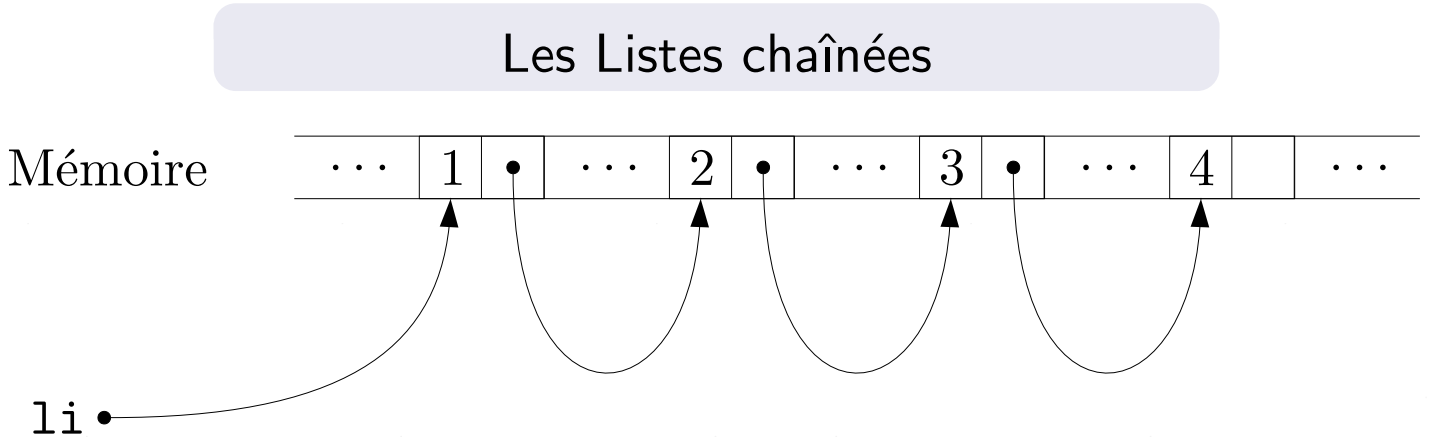
Mémoire

Les Listes chaînées




```
1  let li = [1;2;3;4];;
```

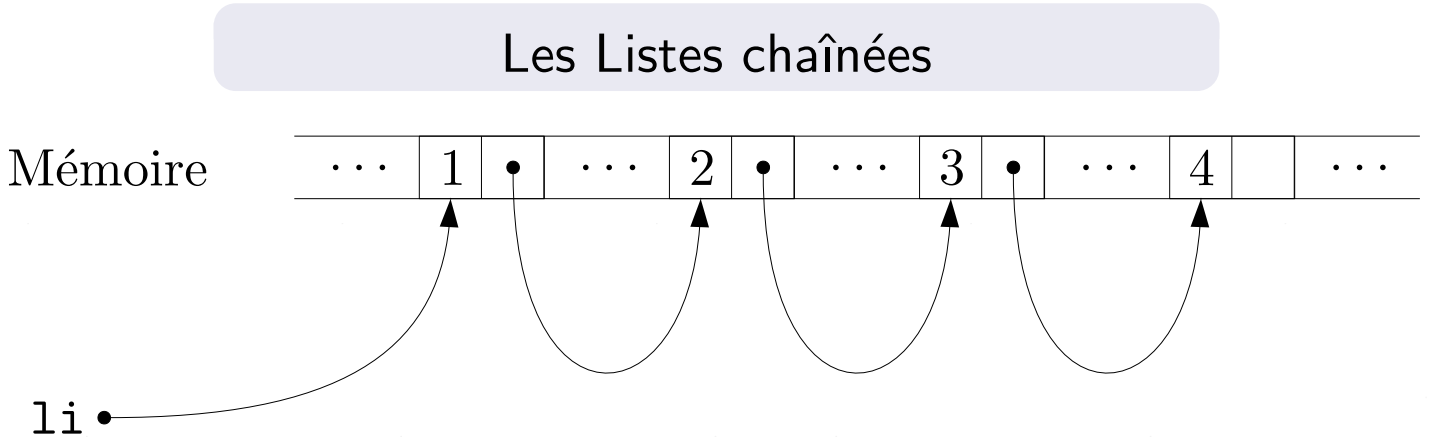
Mémoire




```
1  let li = [1;2;3;4];;
```

- Cases mémoires éloignées

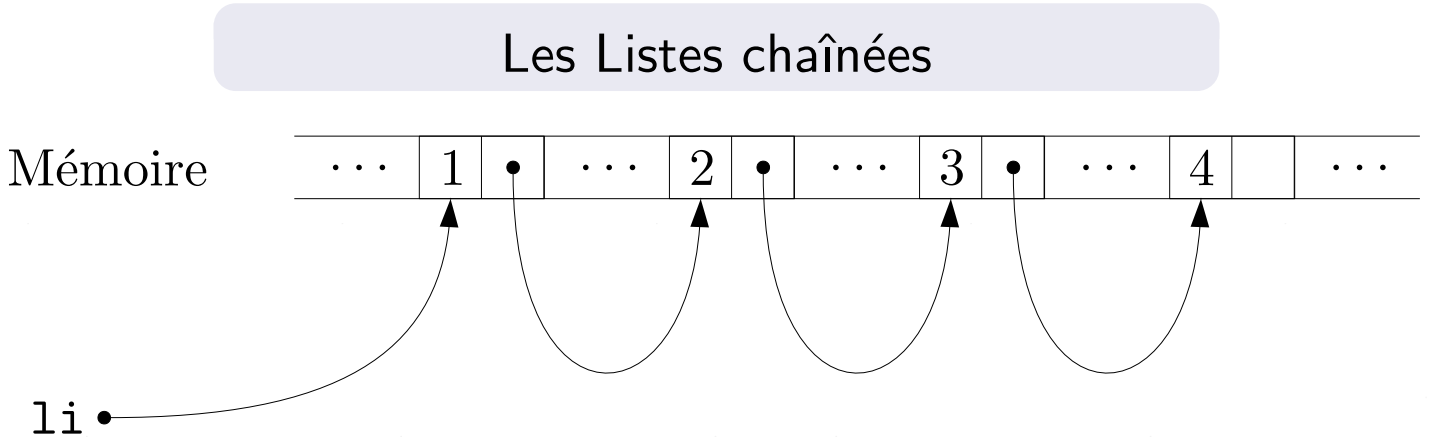
Mémoire




```
1  let li = [1;2;3;4];;
```

- Cases mémoires éloignées
- `li` = référence vers 1er élément

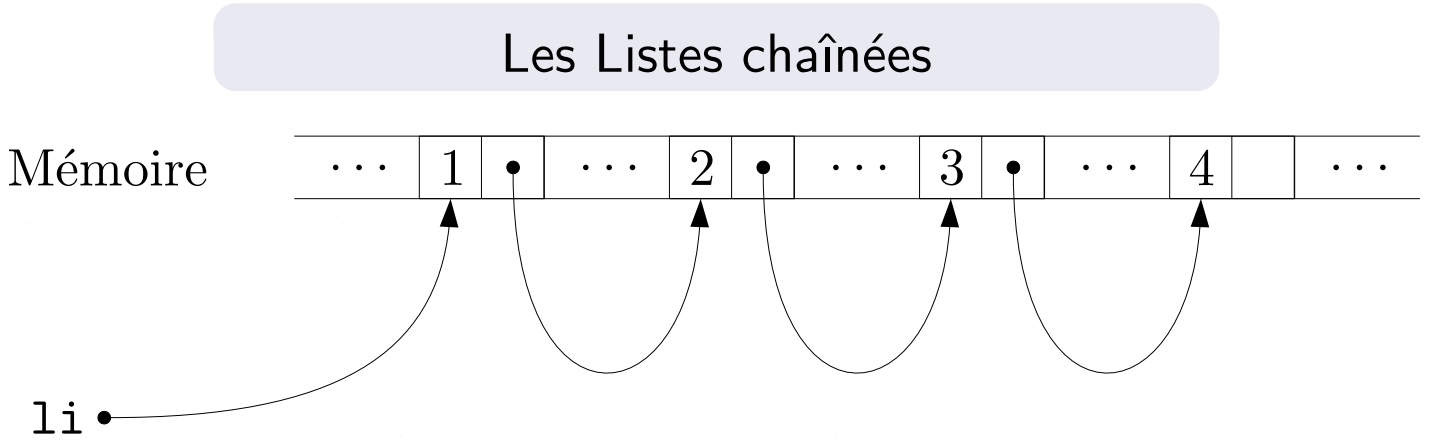
Mémoire




```
1  let li = [1;2;3;4];;
```

- Cases mémoires éloignées
- `li` = référence vers 1er élément
- Chaque élément contient :
  - Une valeur
  - Une référence vers l'élément suivant

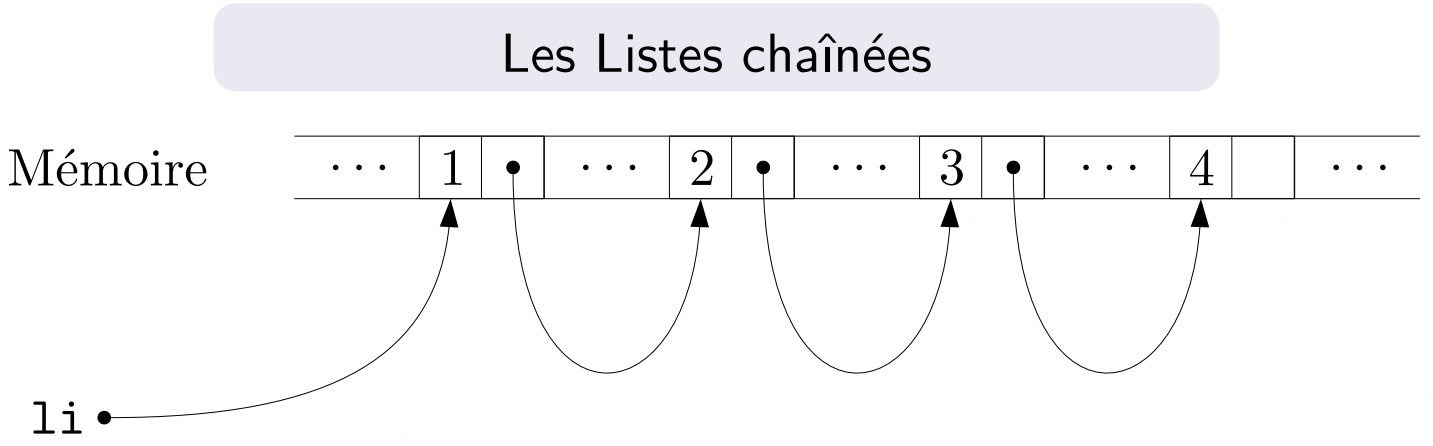
Mémoire




```
1  let li = [1;2;3;4];;
```

- Cases mémoires éloignées
- `li` = référence vers 1er élément
- Chaque élément contient :
  - Une valeur
  - Une référence vers l'élément suivant
- Ajouter/supprimer le 1er élément : rapide

Mémoire




```
1  let li = [1;2;3;4];;
```

- Cases mémoires éloignées
- `li` = référence vers 1er élément
- Chaque élément contient :
  - Une valeur
  - Une référence vers l'élément suivant
- Ajouter/supprimer le 1er élément : rapide
- Accéder au $i^{\text{ème}}$ élément : lent

Sommaire

Le module `List`

- Attention : liste chaînée $\neq$ liste de Python.

Le module `List`

- Attention : liste chaînée $\neq$ liste de Python.
- Définition : une liste de type `'a list` est :
  - Soit `[]` (liste vide)
  - Soit `e :: l` (concaténation d'un élément de type `'a` et d'une liste)

Le module `List`

- Attention : liste chaînée $\neq$ liste de Python.
- Définition : une liste de type `'a list` est :
  - Soit `[]` (liste vide)
  - Soit `e :: l` (concaténation d'un élément de type `'a` et d'une liste)
- `list` est un type récursif.

Le module `List`

- Attention : liste chaînée $\neq$ liste de Python.
- Définition : une liste de type `'a list` est :
  - Soit `[]` (liste vide)
  - Soit `e :: l` (concaténation d'un élément de type `'a` et d'une liste)
- `list` est un type récursif.
- Adaptées pour programmation récursive.

Le module `List`

- Attention : liste chaînée $\neq$ liste de Python.
- Définition : une liste de type `'a list` est :
  - Soit `[]` (liste vide)
  - Soit `e :: l` (concaténation d'un élément de type `'a` et d'une liste)
- `list` est un type récursif.
- Adaptées pour programmation récursive.
- `5::4::8::[]` s'écrit aussi `[5;4;8]`

Le module `List`

- Attention : liste chaînée $\neq$ liste de Python.
- Définition : une liste de type `'a list` est :
  - Soit `[]` (liste vide)
  - Soit `e :: l` (concaténation d'un élément de type `'a` et d'une liste)
- `list` est un type récursif.
- Adaptées pour programmation récursive.
- `5::4::8::[]` s'écrit aussi `[5;4;8]`

Exercice

Écrire une fonction `somme: int list -> int` qui renvoie la somme des éléments d'une liste.

Le module `List`

- Attention : liste chaînée $\neq$ liste de Python.
- Définition : une liste de type `'a list` est :
  - Soit `[]` (liste vide)
  - Soit `e :: l` (concaténation d'un élément de type `'a` et d'une liste)
- `list` est un type récursif.
- Adaptées pour programmation récursive.
- `5::4::8::[]` s'écrit aussi `[5;4;8]`

Exercice

Écrire une fonction `somme: int list -> int` qui renvoie la somme des éléments d'une liste.

```
1  let rec somme: int list -> int = function
2    | []    -> 0
3    | e::q -> e + somme q;;
```

Les fonctions qui suivent sont rarement utilisées.

Les fonctions qui suivent sont rarement utilisées.

- `List.length: 'a list -> int` renvoie la longeur de la liste.

Les fonctions qui suivent sont rarement utilisées.

- `List.length: 'a list -> int` renvoie la longueur de la liste.

```
1  # List.length [3; 6; 9];;
2  - : int = 3
```

Les fonctions qui suivent sont rarement utilisées.

- `List.length: 'a list -> int` renvoie la longueur de la liste.

```
1  # List.length [3; 6; 9];;
2  - : int = 3
```

- `List.hd: 'a list -> 'a` renvoie le 1er élément
  (hd = head = tête de la liste).

Les fonctions qui suivent sont rarement utilisées.

- `List.length: 'a list -> int` renvoie la longueur de la liste.

```
# List.length [3; 6; 9];;
- : int = 3
```

- `List.hd: 'a list -> 'a` renvoie le 1er élément
(hd = head = tête de la liste).

```
# List.hd [1; 2; 3; 4; 5; 6];;
- : int = 1
```

Les fonctions qui suivent sont rarement utilisées.

- `List.length: 'a list -> int` renvoie la longueur de la liste.

```
# List.length [3; 6; 9];;
- : int = 3
```

- `List.hd: 'a list -> 'a` renvoie le 1er élément
(hd = head = tête de la liste).

```
# List.hd [1; 2; 3; 4; 5; 6];;
- : int = 1
```

- `List.tl: 'a list -> 'a list` renvoie la liste sans 1er élément
(tl = tail = queue de la liste).

Les fonctions qui suivent sont rarement utilisées.

- `List.length: 'a list -> int` renvoie la longueur de la liste.

```
# List.length [3; 6; 9];;
- : int = 3
```

- `List.hd: 'a list -> 'a` renvoie le 1er élément
(hd = head = tête de la liste).

```
# List.hd [1; 2; 3; 4; 5; 6];;
- : int = 1
```

- `List.tl: 'a list -> 'a list` renvoie la liste sans 1er élément
(tl = tail = queue de la liste).

```
# List.tl [7.1; 0.4; -1.9];;
- : float list = [0.4; -1.9]
```

Les fonctions qui suivent sont rarement utilisées.

- `List.length: 'a list -> int` renvoie la longueur de la liste.

```
# List.length [3; 6; 9];;
- : int = 3
```

- `List.hd: 'a list -> 'a` renvoie le 1er élément
(hd = head = tête de la liste).

```
# List.hd [1; 2; 3; 4; 5; 6];;
- : int = 1
```

- `List.tl: 'a list -> 'a list` renvoie la liste sans 1er élément
(tl = tail = queue de la liste).

```
# List.tl [7.1; 0.4; -1.9];;
- : float list = [0.4; -1.9]
```

- `(@): 'a list -> 'a list -> 'a list` concatène deux listes

Les fonctions qui suivent sont rarement utilisées.

- `List.length: 'a list -> int` renvoie la longueur de la liste.

```
# List.length [3; 6; 9];;
- : int = 3
```

- `List.hd: 'a list -> 'a` renvoie le 1er élément
(hd = head = tête de la liste).

```
# List.hd [1; 2; 3; 4; 5; 6];;
- : int = 1
```

- `List.tl: 'a list -> 'a list` renvoie la liste sans 1er élément
(tl = tail = queue de la liste).

```
# List.tl [7.1; 0.4; -1.9];;
- : float list = [0.4; -1.9]
```

- `(@): 'a list -> 'a list -> 'a list` concatène deux listes

```
# [1; 2; 3] @ [4; 5; 6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Les accumulateurs

Astuce de programmation

Accumulateur = paramètre supplémentaire d'une fonction où est stocké le résultat des appels récursifs précédents.

Les accumulateurs

Astuce de programmation

Accumulateur = paramètre supplémentaire d'une fonction où est stocké le résultat des appels récursifs précédents.

Exemple

Écrire une fonction `rev: 'a list -> 'a list` qui inverse l'ordre des éléments d'une liste.

Les accumulateurs

Astuce de programmation

Accumulateur = paramètre supplémentaire d'une fonction où est stocké le résultat des appels récursifs précédents.

Exemple

Écrire une fonction `rev: 'a list -> 'a list` qui inverse l'ordre des éléments d'une liste.

Indication : créer d'abord une fonction
`rev_aux: 'a list -> 'a list -> 'a list` telle que :

`rev_aux li acc` renvoie `(rev li) @ acc`

Solution de l'exercice

- **Ne fonctionne pas :**

```
1  (* Ne fonctionne pas *)
2  let rec rev: 'a list -> 'a list = function
3    | []    -> []
4    | e::q -> e::rev q;;
```

- Solution :

```
1  let rec rev_aux (li: 'a list) (acc: 'a list): 'a list =
2    match li with
3    | []    -> acc
4    | e::q -> rev_aux q (e::acc);;
5
6  let rev (li: 'a list): 'a list = rev_aux li [];;
```

Les accumulateurs

Remarque : les accumulateurs permettent également de transformer une boucle `while` en fonction récursive :

```
let fact_it (n: int): int =
  let i = ref n in
  let res = ref 1 in
  while !i > 1 do
    res := !res * !i;
    decr i;     (* i := !i - 1 *)
  done;
  !res;;
```

```
let rec fact_rec_aux (n: int) (i: int) (res: int): int =
  if i <= 1 then res else fact_rec_aux n (i-1) (res*i);;

let fact_rec (n: int): int = fact_rec_aux n n 1;;
```

Sommaire

Le module `Array`

- Adaptés pour programmation itérative

Le module `Array`

- Adaptés pour programmation itérative
- `[|...|]` : construit un tableau

Le module `Array`

- Adaptés pour programmation itérative
- `[|...|]` : construit un tableau

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

Le module `Array`

- Adaptés pour programmation itérative
- `[|...|]` : construit un tableau

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

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

Le module `Array`

- Adaptés pour programmation itérative
- `[|...|]` : construit un tableau

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

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

```
# Array.make 5 1.23;;
- : float array = [|1.23; 1.23; 1.23; 1.23; 1.23|]
```

Le module `Array`

- `t.(i)` : accès à l'élément d'indice `i`

```
# let t = [|1; 4; 7; 10|];;
val t : int array = [|1; 4; 7; 10|]
```

Le module `Array`

- `t.(i)` : accès à l'élément d'indice `i`

```
# let t = [|1; 4; 7; 10|];;
val t : int array = [|1; 4; 7; 10|]
```

```
# t.(0);;
- : int = 1
```

Le module `Array`

- `t.(i)` : accès à l'élément d'indice `i`

```
# let t = [|1; 4; 7; 10|];;
val t : int array = [|1; 4; 7; 10|]
```

```
# t.(0);;
- : int = 1
```

```
# t.(1);;
- : int = 4
```

Le module `Array`

- `t.(i)` : accès à l'élément d'indice `i`

```
# let t = [|1; 4; 7; 10|];;
val t : int array = [|1; 4; 7; 10|]
```

```
# t.(0);;
- : int = 1
```

```
# t.(1);;
- : int = 4
```

```
# t.(2);;
- : int = 7
```

Le module `Array`

- `t.(i)` : accès à l'élément d'indice `i`

```
# let t = [|1; 4; 7; 10|];;
val t : int array = [|1; 4; 7; 10|]
```

```
# t.(0);;
- : int = 1
```

```
# t.(1);;
- : int = 4
```

```
# t.(2);;
- : int = 7
```

```
# t.(3);;
- : int = 10
```

Le module `Array`

- `t.(i)` : accès à l'élément d'indice `i`

```
# let t = [|1; 4; 7; 10|];;
val t : int array = [|1; 4; 7; 10|]
```

```
# t.(0);;
- : int = 1
```

```
# t.(1);;
- : int = 4
```

```
# t.(2);;
- : int = 7
```

```
# t.(3);;
- : int = 10
```

```
# t.(-1);;
Exception: Invalid_argument "index out of bounds".
```

Le module `Array`

- `t.(i) <- e` : modifie l'élément d'indice `i`

```
# let t = [|0; 1; 2; 8; 4; 5|] in
    t.(3) <- 3;
    t;;
- : int array = [|0; 1; 2; 3; 4; 5|]
```

Le module `Array`

- `t.(i) <- e` : modifie l'élément d'indice `i`

```
# let t = [|0; 1; 2; 8; 4; 5|] in
    t.(3) <- 3;
    t;;
- : int array = [|0; 1; 2; 3; 4; 5|]
```

- `Array.length` : taille du tableau

```
# let t = [|5; 5; 5; 5; 5|] in
    Array.length t;;
- : int = 5
```

Le module `Array`

- `t.(i) <- e` : modifie l'élément d'indice `i`

```
# let t = [|0; 1; 2; 8; 4; 5|] in
    t.(3) <- 3;
    t;;
- : int array = [|0; 1; 2; 3; 4; 5|]
```

- `Array.length` : taille du tableau

```
# let t = [|5; 5; 5; 5; 5|] in
    Array.length t;;
- : int = 5
```

Exercice

Écrire une fonction `echange: 'a array -> int -> int -> unit` telle que `echange t i j` échange les éléments d'indice `i` et `j` de `t`.

Solution de l'exercice

```
1  let echange (t: 'a array) (i: int) (j: int): unit =
2    let tmp = t.(i) in
3    t.(i) <- t.(j);
4    t.(j) <- tmp;;
```

Sommaire

Liste Python — Liste chaînée — Tableau

	Liste Python `L`	List Ocaml `li`	Array Ocaml `t`
Paradigme			
Accès aux éléments			
Calcul de la longueur			
Ajout en tête			
Ajout			
Suppression en tête			
Suppression			

Liste Python — Liste chaînée — Tableau

	Liste Python `L`	List Ocaml `li`	Array Ocaml `t`
Paradigme	Itératif ou récursif	Récursif	Itératif
Accès aux éléments			
Calcul de la longueur			
Ajout en tête			
Ajout			
Suppression en tête			
Suppression			

Liste Python — Liste chaînée — Tableau

	Liste Python `L`	List Ocaml `li`	Array Ocaml `t`
Paradigme	Itératif ou récursif	Récursif	Itératif
Accès aux éléments	`L[i]` Temps $\Theta(1)$	`List.nth li i` Temps $\Theta(i)$	`t.(i)` Temps $\Theta(1)$
Calcul de la longueur			
Ajout en tête			
Ajout			
Suppression en tête			
Suppression			

Liste Python — Liste chaînée — Tableau

	Liste Python `L`	List Ocaml `li`	Array Ocaml `t`
Paradigme	Itératif ou récursif	Récursif	Itératif
Accès aux éléments	`L[i]` Temps $\Theta(1)$	`List.nth li i` Temps $\Theta(i)$	`t.(i)` Temps $\Theta(1)$
Calcul de la longueur	`len(L)` Temps $\Theta(1)$	`List.length li` Temps $\Theta(\textit{taille})$	Array.length t Temps $\Theta(1)$
Ajout en tête			
Ajout			
Suppression en tête			
Suppression			

Liste Python — Liste chaînée — Tableau

	Liste Python `L`	List Ocaml `li`	Array Ocaml `t`
Paradigme	Itératif ou récursif	Récursif	Itératif
Accès aux éléments	`L[i]` Temps $\Theta(1)$	`List.nth li i` Temps $\Theta(i)$	`t.(i)` Temps $\Theta(1)$
Calcul de la longueur	`len(L)` Temps $\Theta(1)$	`List.length li` Temps $\Theta(taille)$	Array.length t Temps $\Theta(1)$
Ajout en tête	`L.insert` Temps non constant	`let li2 = e::li` Temps $\Theta(1)$	Pas adapté pour cette opération
Ajout			
Suppression en tête			
Suppression			

Liste Python — Liste chaînée — Tableau

	Liste Python `L`	List Ocaml `li`	Array Ocaml `t`
Paradigme	Itératif ou récursif	Récursif	Itératif
Accès aux éléments	`L[i]` Temps $\Theta(1)$	`List.nth li i` Temps $\Theta(i)$	`t.(i)` Temps $\Theta(1)$
Calcul de la longueur	`len(L)` Temps $\Theta(1)$	`List.length li` Temps $\Theta(taille)$	Array.length t Temps $\Theta(1)$
Ajout en tête	`L.insert` Temps non constant	`let li2 = e::li` Temps $\Theta(1)$	Pas adapté pour cette opération
Ajout	`L.insert,L.append` Temps non constant	Pas adapté pour cette opération	Pas adapté pour cette opération
Suppression en tête			
Suppression			

Liste Python — Liste chaînée — Tableau

	Liste Python `L`	List Ocaml `li`	Array Ocaml `t`
Paradigme	Itératif ou récursif	Récursif	Itératif
Accès aux éléments	`L[i]` Temps $\Theta(1)$	`List.nth li i` Temps $\Theta(i)$	`t.(i)` Temps $\Theta(1)$
Calcul de la longueur	`len(L)` Temps $\Theta(1)$	`List.length li` Temps $\Theta(\textit{taille})$	Array.length t Temps $\Theta(1)$
Ajout en tête	`L.insert` Temps non constant	`let li2 = e::li` Temps $\Theta(1)$	Pas adapté pour cette opération
Ajout	`L.insert,L.append` Temps non constant	Pas adapté pour cette opération	Pas adapté pour cette opération
Suppression en tête	`del L[0]` Temps non constant	`let li2 = List.tl li` Temps $\Theta(1)$	Pas adapté pour cette opération
Suppression			

Liste Python — Liste chaînée — Tableau

	Liste Python `L`	List Ocaml `li`	Array Ocaml `t`
Paradigme	Itératif ou récursif	Récursif	Itératif
Accès aux éléments	`L[i]` Temps $\Theta(1)$	`List.nth li i` Temps $\Theta(i)$	`t.(i)` Temps $\Theta(1)$
Calcul de la longueur	`len(L)` Temps $\Theta(1)$	`List.length li` Temps $\Theta(\textit{taille})$	Array.length t Temps $\Theta(1)$
Ajout en tête	`L.insert` Temps non constant	`let li2 = e::li` Temps $\Theta(1)$	Pas adapté pour cette opération
Ajout	`L.insert,L.append` Temps non constant	Pas adapté pour cette opération	Pas adapté pour cette opération
Suppression en tête	`del L[0]` Temps non constant	`let li2 = List.tl li` Temps $\Theta(1)$	Pas adapté pour cette opération
Suppression	`del L[i]` Temps non constant	Pas adapté pour cette opération	Pas adapté pour cette opération

Choix d'une structure de données

Choix d'une structure de données :

- De quelles opérations ai-je besoin ?

Choix d'une structure de données

Choix d'une structure de données :

- De quelles opérations ai-je besoin ?
  - Ajouter et supprimer des éléments $\rightsquigarrow$ `List`

Choix d'une structure de données

Choix d'une structure de données :

- De quelles opérations ai-je besoin ?
  - Ajouter et supprimer des éléments $\rightsquigarrow$ `List`
  - Modifier les éléments $\rightsquigarrow$ `Array`

Choix d'une structure de données

Choix d'une structure de données :

- De quelles opérations ai-je besoin ?
  - Ajouter et supprimer des éléments $\rightsquigarrow$ `List`
  - Modifier les éléments $\rightsquigarrow$ `Array`

Exemples :

Choix d'une structure de données

Choix d'une structure de données :

- De quelles opérations ai-je besoin ?
  - Ajouter et supprimer des éléments $\rightsquigarrow$ `List`
  - Modifier les éléments $\rightsquigarrow$ `Array`

Exemples :

- J'ai besoin de supprimer les éléments négatifs :
  $\rightsquigarrow$ Utilisation de `List`.

Choix d'une structure de données

Choix d'une structure de données :

- De quelles opérations ai-je besoin ?
  - Ajouter et supprimer des éléments $\rightsquigarrow$ `List`
  - Modifier les éléments $\rightsquigarrow$ `Array`

Exemples :

- J'ai besoin de supprimer les éléments négatifs :
  $\rightsquigarrow$ Utilisation de `List`.
- J'ai besoin d'échanger des éléments :
  $\rightsquigarrow$ Utilisation de `Array`.

Sommaire

Caractères et chaîne de caractères

- `''` : pour définir un caractère

```
# let c = 'a';;
val c : char = 'a'
```

Caractères et chaîne de caractères

- `''` : pour définir un caractère

```
# let c = 'a';;
val c : char = 'a'
```

- `""` : pour définir une chaîne de caractères

```
# let s = "abcde";;
val s : string = "abcde"
```

Caractères et chaîne de caractères

- `''` : pour définir un caractère

```
# let c = 'a';;
val c : char = 'a'
```

- `""` : pour définir une chaîne de caractères

```
# let s = "abcde";;
val s : string = "abcde"
```

- Contrairement à Python : `'a'` $\not\sim$ `"a"`

Caractères et chaîne de caractères

- `''` : pour définir un caractère

```
# let c = 'a';;
val c : char = 'a'
```

- `""` : pour définir une chaîne de caractères

```
# let s = "abcde";;
val s : string = "abcde"
```

- Contrairement à Python : `'a'` $\not\sim$ `"a"`
- `string` $\approx$ `char array`

Caractères et chaîne de caractères

- `''` : pour définir un caractère

```
# let c = 'a';;
val c : char = 'a'
```

- `""` : pour définir une chaîne de caractères

```
# let s = "abcde";;
val s : string = "abcde"
```

- Contrairement à Python : `'a'` $\not\sim$ `"a"`
- `string` $\approx$ `char array`
- `s.[i]` : accès à l'élement d'indice `i`

```
# let s = "abcde" in
      s.[4];;
- : char = 'e'
```

Caractères et chaîne de caractères

- `String.length` : longueur d'une `string`

```
# String.length "Test";;
- : int = 4
```

Caractères et chaîne de caractères

- `String.length` : longueur d'une `string`

```
# String.length "Test";;
- : int = 4
```

- `^` : concaténation de deux `string`

```
# "Hello " ^ "World" ^ " !";;
- : string = "Hello World !"
```

Caractères et chaîne de caractères

- `String.length` : longueur d'une `string`

```
# String.length "Test";;
- : int = 4
```

- `^` : concaténation de deux `string`

```
# "Hello " ^ "World" ^ " !";;
- : string = "Hello World !"
```

- Les `string` sont non-mutables (non modifiables)

```
# s.[0] <- 'A';;
1 | s.[0] <- 'A';;
      ^
Error: This expression has type string but an
       expression was expected of type bytes
```

Sommaire

Matrice

Matrice = tableau de tableaux

Matrice

Matrice = tableau de tableaux

```
# let m = Array.make_matrix 3 4 0;;
val m : int array array = [|[|0; 0; 0; 0|]; [|0; 0; 0;
    0|]; [|0; 0; 0; 0|]|]
```

Matrice

Matrice = tableau de tableaux

```
# let m = Array.make_matrix 3 4 0;;
val m : int array array = [|[|0; 0; 0; 0|]; [|0; 0; 0; 0|]; [|0; 0; 0; 0|]|]
```

```
# m.(0).(1) <- 9;;
- : unit = ()
# m;;
- : int array array = [|[|0; 9; 0; 0|]; [|0; 0; 0; 0|]; [|0; 0; 0; 0|]|]
```

Matrice

Matrice = tableau de tableaux

```
# let m = Array.make_matrix 3 4 0;;
val m : int array array = [|[|0; 0; 0; 0|]; [|0; 0; 0;
    0|]; [|0; 0; 0; 0|]|]
```

```
# m.(0).(1) <- 9;;
- : unit = ()
# m;;
- : int array array = [|[|0; 9; 0; 0|]; [|0; 0; 0; 0|];
    [|0; 0; 0; 0|]|]
```

Attention, ne pas utiliser `Array.make 3 (Array.make 4 0)` :

```
# let m = Array.make 3 (Array.make 4 0) in
    m.(0).(0) <- 1;
    m;;
- : int array array = [|[|1; 0; 0; 0|]; [|1; 0; 0; 0|];
    [|1; 0; 0; 0|]|]
```

Sommaire

`List.map`

Hors programme

`List.map`

Hors programme

- `List.map: ('a -> 'b) -> 'a list -> 'b list`
- `List.map f [a1; a2; ...; an]` s'évalue en `[f a1; f a2; ...; f an]`

List.map

Hors programme

- `List.map: ('a -> 'b) -> 'a list -> 'b list`
- `List.map f [a1; a2; ...; an]` s'évalue en `[f a1; f a2; ...; f an]`

Exercice

Écrire une fonction qui prend en entrée une liste d'entiers et élève ses éléments au carré.

List.map

Hors programme

- `List.map: ('a -> 'b) -> 'a list -> 'b list`
- `List.map f [a1; a2; ...; an]` s'évalue en `[f a1; f a2; ...; f an]`

Exercice

Écrire une fonction qui prend en entrée une liste d'entiers et élève ses éléments au carré.

```
let carre (li: int list): int list =
  List.map (fun x -> x*x) li;;
```

```
# carre [1; 2; 3; 4];;
- : int list = [1; 4; 9; 16]
```

List.iter

Hors programme

List.iter

Hors programme

- `List.iter: ('a -> unit) -> 'a list -> unit`
- `List.iter f [a1; a2; ...; an]` effectue `f a1; f a2; ...; f an`

List.iter

Hors programme

- `List.iter: ('a -> unit) -> 'a list -> unit`
- `List.iter f [a1; a2; ...; an]` effectue `f a1; f a2; ...; f an`

Exercice

Écrire une fonction qui affiche les éléments d'une liste d'entiers.

List.iter

Hors programme

- `List.iter: ('a -> unit) -> 'a list -> unit`
- `List.iter f [a1; a2; ...; an]` effectue `f a1; f a2; ...; f an`

Exercice

Écrire une fonction qui affiche les éléments d'une liste d'entiers.

```
1  let print_list (li: int list): unit =
2    let f x = print_int x;
3              print_char ' ' in
4    List.iter f li;;
```

List.iter

Hors programme

- `List.iter: ('a -> unit) -> 'a list -> unit`
- `List.iter f [a1; a2; ...; an]` effectue `f a1; f a2; ...; f an`

Exercice

Écrire une fonction qui affiche les éléments d'une liste d'entiers.

```
let print_list (li: int list): unit =
  let f x = print_int x;
            print_char ' ' in
  List.iter f li;;
```

```
# print_list [1; 2; 3; 4];;
1 2 3 4 - : unit = ()
```

List.iter

Hors programme

- `List.iter: ('a -> unit) -> 'a list -> unit`
- `List.iter f [a1; a2; ...; an]` effectue `f a1; f a2; ...; f an`

Exercice

Écrire une fonction qui affiche les éléments d'une liste d'entiers.

```
let print_list (li: int list): unit =
  let f x = print_int x;
            print_char ' ' in
  List.iter f li;;
```

```
# print_list [1; 2; 3; 4];;
1 2 3 4 - : unit = ()
```

- Pour ceux qui connaissent `printf` (comme en C) :

```
let print_list li = List.iter (Printf.printf "%d ") li;;
```