CAML

4 - Les listes

http://tsi.tuxfamily.org/OCaml

24 février 2024

Les tableaux (que l'on verra plus tard)

17	8	75	21	7	3

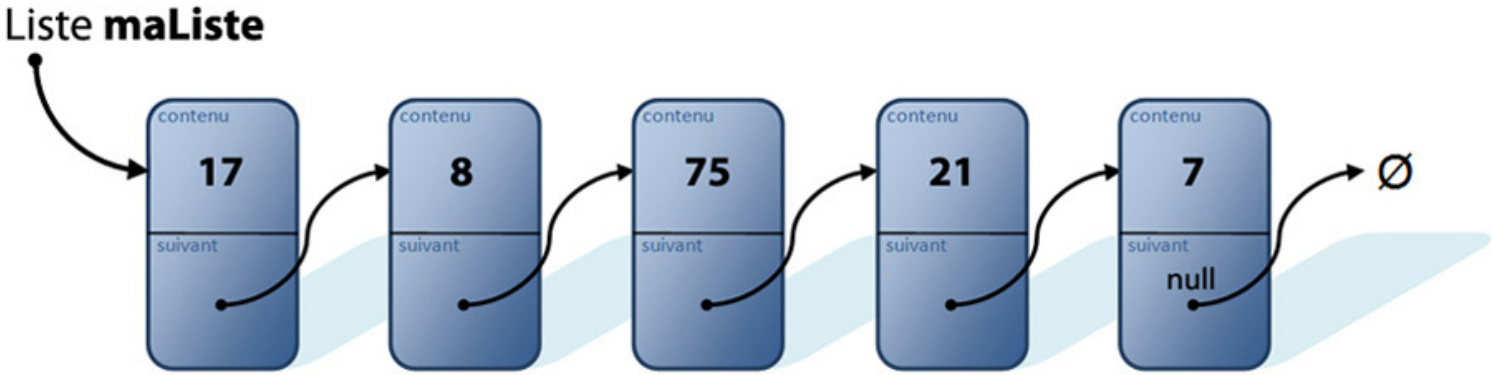
Avantages :

- Accès simple rapide à l'élément de rang n

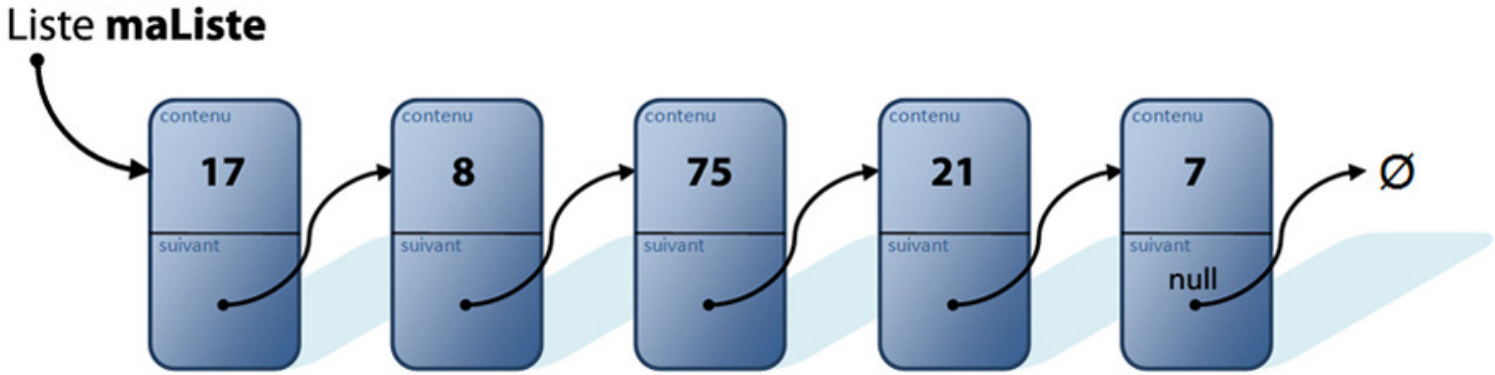
Inconvénients :

- Taille fixe.

Les listes sont des **listes chaînées**



Les listes sont des **listes chaînées**



Avantages :

- On peut modifier la taille.
- On peut "facilement" insérer ou supprimer un élément.

Inconvénient :

- Pour atteindre l'élément d'indice k, il faut parcourir la liste.

Les noms des fonctions seront ici précédés de `List`.

Si on ne souhaite pas préciser `List.fonction` à chaque utilisation d'une fonction, on peut ouvrir le module en début de programme et directement appeler `fonction`.

```
open List;;
```

dans notre présentation, nous écrirons toujours `List.fonction`

Vocabulaire

On dispose de deux **constructeurs** pour créer une liste

Vocabulaire

On dispose de deux **constructeurs** pour créer une liste

- Le constructeur [] permet de créer une liste vide. *On lit "nil"*

Vocabulaire

On dispose de deux **constructeurs** pour créer une liste

- Le constructeur `[]` permet de créer une liste vide. *On lit "nil"*
- Le constructeur `::` permet d'ajouter un élément en tête de liste.*On lit "conse" ou "4 points"*

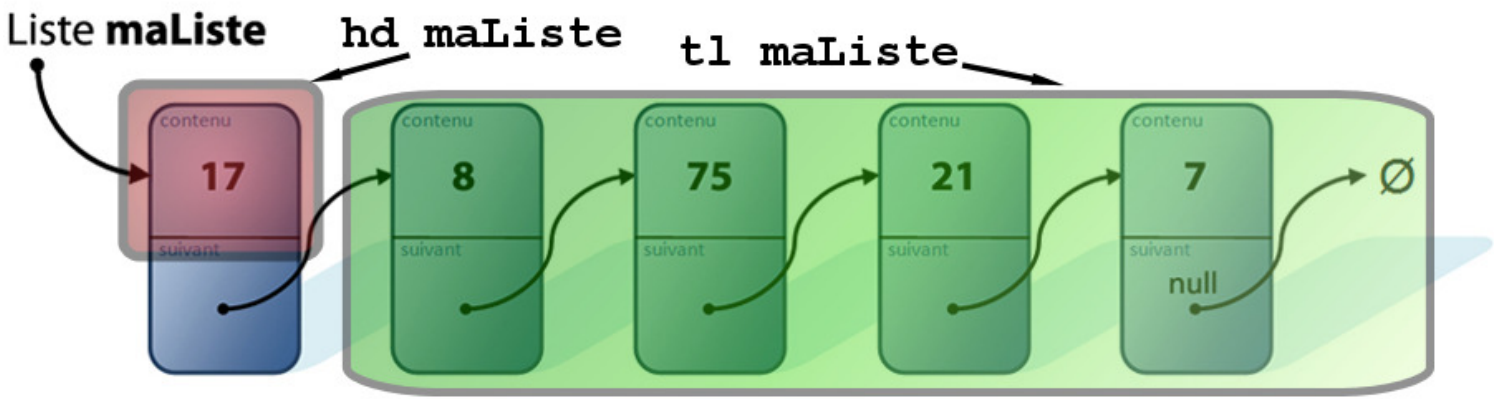
```
# let vide = [];;
val vide : 'a list = []
# let pairs = [0;2;4;6;8];;
val pairs : int list = [0; 2; 4; 6; 8]
# let impairs = 1::3::5::7::[];;
val impairs : int list = [1; 3; 5; 7]
```

Vocabulaire

On dispose de **selecteurs** pour accéder aux éléments d'une liste

- Le sélecteur `hd` permet d'accéder à l'élément de tête de liste (premier élément). *On lit "head"*

```
# List.hd [17;8;75;21;7];;
- : int = 17
```

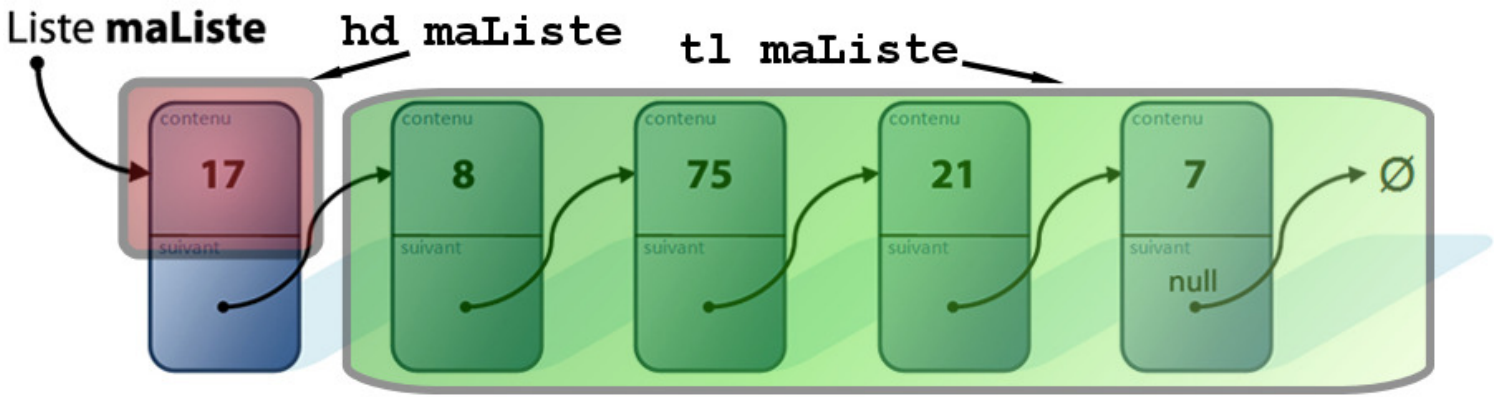


Vocabulaire

On dispose de **selecteurs** pour accéder aux éléments d'une liste

- Le sélecteur `tl` retourne la queue de liste..*On lit "tail"*

```
# List.tl [17;8;75;21;7];;
- : int list = [8; 75; 21; 7]
```



Les éléments d'une liste ne sont pas modifiables.

```
# let rec afficher = function
        | [] -> ()
        | h :: t -> print_int h;afficher t
;;
val afficher : int list -> unit = <fun>
# afficher [1;4;5];;
145- : unit = ()
```

```
# let rec afficher = function
        | [] -> ()
        | h :: t -> print_int h;afficher t
;;
val afficher : int list -> unit = <fun>
# afficher [1;4;5];;
145- : unit = ()
```

Ce filtrage est exhaustif car une liste est

- Soit vide `[]`
- Soit de la forme `h :: t` où
  - `h` est la tête (un élément)
  - `t` la queue (une liste)

La fonction `List.length` renvoie la longueur d'une liste

La fonction `List.length` renvoie la longueur d'une liste

Exercice 1

Reprogrammer cette fonction à l'aide d'une fonction récursive et d'un filtrage :

```
let rec long = function
    | [] -> ...
    | h::t -> ...
;;
```

Solution 1 :

```
let rec long = function
        | [] -> 0
        | h::t -> 1 + long t;;
```

Solution 1 :

```
let rec long = function
        | [] -> 0
        | h::t -> 1 + long t;;
```

Solution 2 : solution récursive terminale

```
let long l =
        let rec longR n = function
                | [] -> n
                | h::t -> longR (n+1) t
        in longR 0 l;;
```

La fonction `List.rev` renvoie une liste "miroir" d'une liste donnée. (Les éléments sont en ordre inverse)

La fonction `List.rev` renvoie une liste "miroir" d'une liste donnée. (Les éléments sont en ordre inverse)

Exercice 2

Reprogrammer cette fonction à l'aide d'une fonction récursive terminale et d'un filtrage

Voici une solution :

```
let miroir l =
        let rec miroirR acc = function
                | [] -> acc
                | h::t -> miroirR (h::acc) t
        in miroirR [] l
;;
```

Le constructeur `@` concatène deux listes.

Vocabulaire

Comme pour `::`, l'opérateur `@` est un opérateur **infixe** :

```
liste1 @ liste2
```

Remarque : La fonction suivante produit le même effet, mais s'utilise de manière préfixée :

```
List.append : 'a list -> 'a list -> 'a list
```

Exercice 3

1. Reprogrammer cette fonction de manière récursive.
2. Adapter la fonction précédente en une version récursive terminale

Version 1 :

```
let rec concat l1 l2 = match l1 with
        | [] -> l2
        | (h::t) -> h::(concat t l2);;
```

Version 2 : récursive terminale

```
let concat l1 l2 =
        let rec concatR l = function
        | [] -> l
        | (h::q) -> concatR (h::l) q
   in concatR l2 (List.rev l1);;
```

Les temps d'exécution de ces deux fonctions sont à peu près identiques et bien moins rapides que la fonction `@` qui se contente de faire pointer la fin de la première liste sur le début de la deuxième. **Dans tous les cas**, la complexité de l'opération `L1 @ L2` est en $O(|\texttt{L1}|)$

`List.iter : ('a -> unit) -> 'a list -> unit` : Effectue une fonction de type `'a -> unit` sur tous les éléments d'une liste.

```
# List.iter print_char ['a';'e';'i';'o';'u'];;
aeiou- : unit = ()
```

Exercice 5

Écrire à l'aide de la fonction `iter` une fonction `carres` qui affiche les carrés d'une liste ainsi :

```
#carres [3;5;9;0];;
3^2=9 / 5^2=25 / 9^2=81 / 0^2=0 / - : unit = ()
```

Exercice 5

Écrire à l'aide de la fonction `iter` une fonction `carres` qui affiche les carrés d'une liste ainsi :

```
#carres [3;5;9;0];;
3^2=9 / 5^2=25 / 9^2=81 / 0^2=0 / - : unit = ()
```

```
let carres l =
        let aff x =
                print_int x;
                print_string "^2=";
                print_int (x*x);
                print_string " / "
        in List.iter aff l;;

carres [3;5;9;0];;
```

`map : ('a -> 'b) -> 'a list -> 'b list` : Renvoie une liste où les éléments sont les images des éléments de la liste de départ par la fonction donnée.

```
# let l = [1;3;5];;
val l : int list = [1; 3; 5]
# List.map (function x-> x*x) l;;
- : int list = [1; 9; 25]
# l;;
- : int list = [1; 3; 5]
```

Il s'agit d'une **nouvelle** liste : la liste de départ n'est pas modifiée.