Constructeurs et accesseurs pour le type `list`

- constructeur de liste vide est `[]` au lieu de `Nil` pour le type `mylist`.
- `constructeur` permettant de rajouter un élément à une liste est `::` mais contrairement au constructeur `C` du type `mylist`, il s'utilise en notation infixe.

  Ainsi on écrit `e :: l` au lieu de `C(e,l)` dans le type `mylist`.

Les `accesseurs` associés à ce constructeur sont `List.hd` et `List.tl` pour récupérer respectivement la tête (head) `e` et la queue (tail) `l` de la liste `e :: l`.
Ces accesseurs sont peu utilisés du fait que l'on utilisera le plus souvent la construction `match` pour déconstruire une liste.

La fonction de calcul de la longueur d'une liste qui s'écrivait avec le type `mylist`

```
type 'a mylist = Nil | C of 'a * 'a mylist
let rec mylist_length l =
    match l with
      Nil -> 0
    | C(_, t) -> 1 + mylist_length t
```

s'écrit comme suit avec le type prédéfini et la construction `match`

```
let rec list_length l =
    match l with
      [] -> 0
    | _ :: t -> 1 + list_length t
```

ou encore sans utiliser `match`

```
 let rec list_length l =
  if l = [] then 0
  else 1 + list_length (List.tl l)
```

Format externe des listes

La notation `e1 :: e2 :: e3 ... :: en :: []` n'étant pas très agréable à lire, `OCaML` utilise un *format externe* pour écrire et lire des listes: `[e1; e2; ...; en]` est le format sous lequel `OCaML` affiche une liste et un format que l'utilisateur peut utiliser pour entrer une liste.

Exemples:

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
# [1.; 2.; 3.];;
- : float list = [1.; 2.; 3.]
# List.hd [1; 2; 3];;
- : int = 1
# List.tl [1; 2; 3];;
- : int list = [2; 3]
# 3 * 3 :: [1; 2; 3];;
- : int list = [9; 1; 2; 3]
# let x = 4;;
val x : int = 4
```

Concaténation de listes

La fonction prédéfinie `List.append l1 l2` retourne une liste constituée des éléments de `l1` suivis des éléments de `l2`.

Exemples:

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

Il existe une version infixe de cette fonction: l'opérateur `@`.

Exemples:

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

Concaténation

à utiliser avec parcimonie.
En effet, sa complexité est en $O(len(l_1))$ car il entraîne une copie de la liste `l1`.
Il peut servir ponctuellement à ajouter[6] un élément `e` en fin d'une liste `l` en utilistant l'expression `l @ [e]`
Par contre l'ajout d'un élément `e` en tête d'une liste `l` se fait en temps constant $O(1)$ grâce à l'expression: `e :: l`.
Le plus souvent, il est préférable de construire une liste à l'envers puis de la retourner en utilisant la fonction `List.rev l` (linéaire par rapport à la longueur de la liste).

[6] en fait, construire une nouvelle liste ayant les mêmes éléments que `l` et `e` à la fin

Exemples de fonctions simples sur les listes

- `List.length`
- `List.mem`
- `List.append`
- `List.rev`
- `List.sort`
- `List.filter`
- `List.map`
- `List.find`, `List.find_all`
- ...

Et il y en a d'autres:

- utiliser `List.` *complétion* pour voir leur nom
- taper leur nom pour voir leur type

type `option` prédéfini

```
type 'a option = Some of 'a | None
```

type de retour uniforme pour les fonctions ne retournant pas toujours une valeur comme les fonctions de recherche par exemple.

Exemples en direct

```
let rec find_if pred l =
  match l with
    [] -> None
  | e :: t -> if pred e then Some e
              else find_if pred t
```

Comparaison pratique

- utiliser `Sys.time` (appel à la command Unix `time`)
- faire la différence entre les temps de fin et de début de l'excécution

```
utop[30]> Sys.time();;
- : float = 6.536402
utop[31]> let start = Sys.time();;
val start : float = 6.540147
utop[32]> Sys.time() -. start;;
- : float = 0.00526800000000005
```

Exemple de comparaison pratique

```
let time f =
  let start = Sys.time() in
  let _ = f () in
  Sys.time() -. start
```

permet de mesurer le temps d'exécution de la fonction `f` sans argument passée en paramètre.

```
utop[36]> time;;
- : (unit -> 'a) -> float = <fun>
utop[37]> time (fun () -> List.mem 100 (iota 100));;
- : float = 8.00000000111822374e-06
```

Exemples avec trois versions de la fonction qui renverse une liste.
(voir `test-efficacite.ml`)