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`)

Fonction `List.fold_left`

```
utop[6]> List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

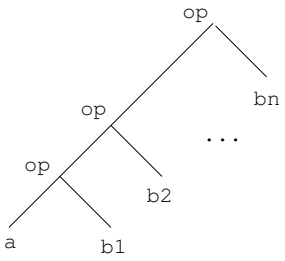
`List.fold_left op a l`

`op` est un opérateur binaire `'a -> 'b -> 'a`

`a` est une valeur de type `'a`

`l` une liste d'éléments de type `'b` : `[b1; b2; ...; bn]`

calcule `op (op ... (op (op a b1) b2) ... ) bn`



Fonction `List.fold_right`

```
utop[7]> List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

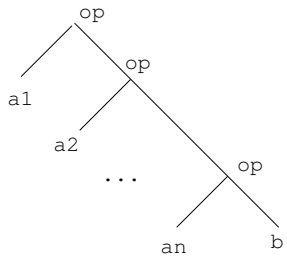
`List.fold_right op l b`

`op` est un opérateur binaire `'a -> 'b -> 'b`

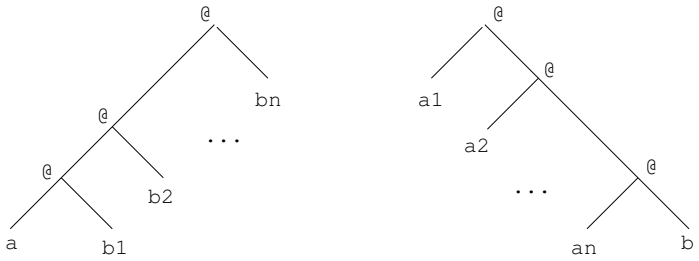
`b` est une valeur de type `'b`

`l` une liste d'éléments de type `'a` : `[a1; a2; ...; an]`

calcule `op a1 (op a2 (... (op an b))...)`



`List.fold_right` vs `List.fold_left`



Comparaison théorique: notion de complexité

estimer théoriquement l'efficacité d'une fonction

- efficacité en temps (nombre d'opérations élémentaires)
- efficacité en espace (espace alloué)
- temps $>=$ espace

Notation $\mathcal{O}$:
La fonction f est dite en $\mathcal{O}(g)$ ssi

$$\exists k \in \mathbb{N}, \exists c > 0, \forall n > k, f(n) \leq cg(n)$$

Exemple: $\mathcal{O}(n \mapsto log_2(n))$
Par abus de notation: $\mathcal{O}(log_2(n))$

Exemples: $\mathcal{O}(log_2(n))$, $\mathcal{O}(n)$, $\mathcal{O}(n^2)$, ...

Comparaison théorique: exemple

```
let rec append l1 l2 =
  match l1 with
    [] -> l2
  | h :: t -> h :: append t l2
```

1. Combien d'appels récursifs de `append` ?
2. Combien de fois l'opérateur `::` est-il utilisé?

La récursion se faisant sur `l1`, la complexité dépend uniquement de la longueur de la liste `l1`.
Soit $a(n)$ le nombre d'appels récursifs pour `l1` de longueur n.

$$\begin{cases} a(0)=0 \\ a(n)=1+a(n-1) \end{cases}$$

se résout en $a(n) = n$.
le nombre d'appel à `::` est égal au nombre d'appels récursifs.
La fonction est en $\mathcal{O}(len(l_1))$.

Comparaison théorique: reverse quadratique

```
let rec reverse l =
  match l with
    [] -> []
  | h :: t -> append (reverse t) [h]
```

Soit $c(n)$ le nombre d'utilisation de `::` lors d'un appel à `reverse l` pour une liste `l` de longueur n.

$$\begin{cases} c(0)=0 \\ c(n)=a(n-1)+c(n-1) \end{cases}$$

se résout en $n(n-1)/2$.
La fonction est en $\mathcal{O}(len(l)^2)$.

Comparaison théorique: reverse linéaire

```
let rec rev_append l acc = (* O(len(l)) *)
  match l with
    [] -> acc
    | h :: t -> rev_append t (h :: acc)

let reverse l = rev_append l []
```

On peut montrer que le nombre d'appels récursifs est égal à n si n est la longueur de la liste `l`.
La fonction est en $\mathcal{O}(len(l))$.