Fonctions d'ordre supérieur

I. `(+un x)`, `(+un-a-tous ls)`, `(map proc ls)`, `(add-a-tous x ls)`

```
(define +un (lambda (x) (+ x 1)))

(define +un-a-tous
  (lambda (ls)
    (if (null? ls) '() (cons (+ 1 (car ls)) (+un-a-tous (cdr ls))))))

(define map
  (lambda (proc ls)
    (if (null? ls)
        '()
        (cons (proc (car ls))
              (map proc (cdr ls))))))
(map +un '(1 2-8i 3.2)) => (2 3-8i 4.2) ; même que +un-a-tous
```


Fonctions d'o.s.

```
(define add-a-tous
  (lambda (x ls)
    (if (null? ls)
        '()
        (cons (+ x (car ls))
              (add-a-tous x ls)))))

(map (lambda (y) (+ x y)) ls) => même que (add-a-tous x ls)
```

Currification

def : soit une fonction f à $n > 1$ paramètres, la forme currifiée de f est une fonction f' à un paramètre (le premier paramètre de f) qui retourne comme résultat une fonction currifiée traitant les paramètres restants.

```
(define curry+
; c'est une fonction à un argument [x] qui
; retourne une fonction à un argument [y] qui retourne x+y
  (lambda (x)
    (lambda (y) (+ x y))))
(curry+ 1) => même que +un
((curry+ 4) 5) => 9
(map (curry+ x) ls) => même que (add-a-tous x ls)

(define cons-avec-car
  (lambda (a)
    (lambda (d) (cons a d))))
(cons-avec-car 1) => (lambda (d) (cons 1 d))
((cons-avec-car 2) '(a b)) => (2 a b)
```


Currification II

(comment currifier une fonction récursive ?)

```
(define map-c
  (lambda (proc)
      (letrec
      ; letrec est obligatoire pour introduire
      ; une fonction «locale» recursive
        ((bob (lambda (ls)
                (if (null? ls)
                    '()
                    (cons (proc (car ls))
                          (bob (cdr ls)))))))
        bob ; c'est le corps de letrec: s'évalue à une fonction récursive
       )))
(map-c +un) => même que +un-a-tous
((map-c car) '((1 2) (3 5) (a))) => (1 3 a)
```

Currification III

```
(define add-a-tous-c
  (lambda (x)
      (letrec
        ((bob (lambda (ls)
                (if (null? ls)
                    '()
                    (cons (+ x (car ls))
                          (bob (cdr ls)))))))
         bob
        )))
(add-a-tous-c 1) => même que +un-a-tous
((add-a-tous 8) '(2 3)) => (10 11)
```


Abstraction procédurale

récursion simple sur une liste en général :

```
(lambda (ls) ; c'est la fonction ce que l'on veut construire
  (if (null? ls)
        ... ; valeur de la fonction sur la liste vide
        ... ; sinon: recursion avec une fonction de type (lambda (a res) ...)
            ; où a est le car est res est le résulta récursif avec le cdr
   ))
; solution:
(define recursion
  (lambda (germe combine)
    (letrec ((bob
              (lambda (ls)
                (if (null? ls)
                    germe
                    (combine
                       (car ls)
                       (bob (cdr ls)))))))
        bob )))
```

Abstraction procédurale II

```
(recursion '() (lambda (a res) (cons (+un a) res)))  => +un-a-tous

(define map-c2 ; définition altérnative de map-c
  (lambda (proc)
    (recursion '()
               (lambda (a res)
                   (cons (proc a) res)))))

(define add-a-tous-c2 ; définition altérnative de add-a-tous-c
  (lambda (x)
    (recursion
      '()
      (lambda (a res) (cons (+ x a) res)))))

(define produit ; produit de tous les éléments d'une liste
  (recursion 1 *))

(define somme ; somme de tous les éléments d'une liste
  (recursion 0 +))
```