



PROGRAMMATION D'ORDRE SUPÉRIEUR

Fonctions en argument

Fonctions en résultat

Abstraction

Map

Apply

RETOUR AU B.A. BA

Fonction Somme

```
(define somme ; → nb
  (lambda (L); liste nb
    (if (null? L)
        0
        (+ (car L)
           (somme (cdr L))))))
```

Fonction Produit

```
(define produit ; → nb
  (lambda (L); liste nb
    (if (null? L)
        1
        (* (car L)
           (produit (cdr L))))))
```

ESSAYONS DE GÉNÉRALISER

```
(define CalculSurListe ; → nombre
  (lambda (L ) ; L liste de nombres
    (if (null? L)
        ValeurSiVide ; élément neutre
        (FonctionDeCalcul (car L)
                          (CalculSurListe (cdr L)..))))))
```

PASSER UNE FONCTION EN ARGUMENT

- En Scheme, il est possible d'utiliser des arguments de type « fonction »
- Exemple : une fonction qui
 - étant donné une fonction et un argument
 - applique deux fois la fonction à l'argument

FONCTION APPLIQUE2FOIS

```
(define applique2fois ; → résultat de f
  (lambda (f n) ; f fonction, n argt de f
    (f (f n))))
```

- Exemples :
(applique2fois sqrt 16) → 2
(applique2fois cdr '(a b c d)) → (c d)
- Comment mémoriser la fonction
« applique2fois sqrt » ?

FONCTION EN RÉSULTAT (1)

- On voudrait écrire :
(define doubleRacine
 (DoublerFonction sqrt))
- Pour ensuite utiliser :
(doubleRacine 16) → 2
- Il faut donc définir la fonction DoublerFonction,
qui retourne une fonction

FONCTION EN RÉSULTAT (2)

- Qu'est ce qu'une fonction ?
 - Une lambda-expression
- Une fonction qui retourne une fonction retourne donc une lambda-expression

```
(define DoublerFonction ; → fonction  
  (lambda (f) ; f fonction à répéter  
    (lambda (n) ; résultat de  
      (f (f n))) )) ; DoublerFonction
```

AUTRE EXEMPLE : GÉNÉRER UNE FONCTION D'ADDITION

```
(define add-gen ; → fonction qui ajoute  
                x à un nombre  
  (lambda (x) ; x nombre  
    (lambda (y)  
      (+ x y))))
```

- (define add-5 (add-gen 5))
- (add-5 9) → 14
- ((add-gen 5) 3) → 8

ABSTRACTION

- Reprenons notre fonction pour généraliser les calculs sur une liste de nombres

```
(define CalculSurListe ; → nombre
  (lambda (L f n) ; L liste de nombres,
                f fonction de 2 nb, n nombre
    (if (null? L)
        n
        (f (car L) (CalculSurListe (cdr L) f n))))))
```

UTILISATION DE LA FONCTION ABSTRAITE

```
(define somme ; → nombre  
  (lambda (L) ; L liste de nombres  
    (CalculSurListe L + 0)))
```

```
(define produit ; → nombre  
  (lambda (L) ; L liste de nombres  
    (CalculSurListe L * 1)))
```

UN AUTRE EXEMPLE D'ABSTRACTION (1)

- Une fonction qui ajoute x à tous les éléments d'une liste de nombres

```
(define ajoute-x ; → liste de nombres
  (lambda (x L) ; x nb, L liste de nb
    (if (null? L)
        '()
        (cons (+ x (car L))
              (ajoute-x x (cdr L))))))
```

UN AUTRE EXEMPLE D'ABSTRACTION (2)

- Une fonction qui multiplie par 2 tous les éléments d'une liste de nombres

```
(define double ; → liste de nombres
  (lambda (L) ; L liste de nb
    (if (null? L)
        '()
        (cons (* 2 (car L))
              (double (cdr L))))))
```

UN AUTRE EXEMPLE D'ABSTRACTION (3)

- Généralisation : appliquer une fonction à tous les éléments d'une liste

```
(define applique-à-tous ; → liste
  (lambda (f L) ; f fonction unaire, L liste
    (if (null? L)
        '()
        (cons (f (car L))
              (applique-à-tous f (cdr L))))))
```

UN AUTRE EXEMPLE D'ABSTRACTION (4)

```
(define carres ; → liste de nb  
  (lambda (l) ; liste de nb  
    (applique-à-tous sqr l)))
```

UN AUTRE EXEMPLE D'ABSTRACTION (5)

```
(define ajoute-x ; → liste de nombres  
  (lambda (x L) ; x nb, L liste de nb  
    (applique-à-tous  
      (lambda (y) (+ y x)) L)))
```

```
(define double ; → liste de nombres  
  (lambda (L) ; L liste de nb  
    (applique-à-tous  
      (lambda (x) (* x 2)) L)))
```

ABSTRACTION DES PARCOURS D'ARBRES (1)

```
(define somme ; → nombre
  (lambda (A) ; A arbre de nb
    (if (vide? A)
        0
        (+ (valeur A)
           (somme (fils-g A))
           (somme (fils-d A))))))
```

ABSTRACTION DES PARCOURS D'ARBRES (2)

```
(define produit ; → nombre
  (lambda (A) ; A arbre de nb
    (if (vide? A)
        1
        (* (valeur A)
           (produit (fils-g A))
           (produit (fils-d A))))))
```

ABSTRACTION DES PARCOURS D'ARBRES (3)

```
(define absArbres ; → nombre
  (lambda (f n A) ; A arbre de nb,
    n nombre, f fonction ternaire
    (if (vide? A)
        n
        (f (valeur A)
            (absArbres f n (fils-g A))
            (absArbres f n (fils-d A))))))
```

ABSTRACTION DES PARCOURS D'ARBRES (4)

```
(define somme ; → nombre  
  (lambda (A) ; A arbre de nb  
    (absArbre + 0 A)))
```

```
(define produit ; → nombre  
  (lambda (A) ; A arbre de nb  
    (absArbre * 1 A)))
```

ABSTRACTION DES PARCOURS D'ARBRES (5)

```
(define maximum ; → nombre  
  (lambda (A) ; A arbre de nb positifs  
    (absArbre max 0 A)))
```

```
(define compter ; → nombre  
  (lambda (A) ; A arbre  
    (absArbre  
      (lambda (x y z) (+ 1 y z))  
      0  
      A)))
```

LA FONCTION MAP

- La fonction applique-à-tous est tellement utile qu'elle est prédéfinie en Scheme
- Elle porte le nom `map` et est en fait une version plus générale de applique-à-tous

EXEMPLES (1)

- Map s'utilise comme applique-à-tous
- `(map (lambda (x) (+ x 10)) '(10 3 100 9 64))`
→ `(20 13 110 19 74)`
- `(map number? '(10 z e (1 2) 3 g 4))`
→ `(#t #f #f #f #t #f #t)`
- `(map (lambda (z) (cons z '(k))) '(1 2 a b))`
→ `((1 k) (2 k) (a k) (b k))`

EXEMPLES (2)

- Map est plus générale qu'applique-à-tous car la fonction à appliquer peut prendre plus d'un argument
- Il faut alors donner à cette fonction autant de listes qu'elle doit avoir d'arguments, toutes les listes devant être de même longueur
- $(\text{map } (\text{lambda } (a \ b) \ (+ \ a \ b)) \ '(1 \ 2 \ 3) \ '(10 \ 20 \ 30)) \rightarrow (11 \ 22 \ 33)$
- $(\text{map } + \ '(1 \ 2 \ 3) \ '(10 \ 20 \ 30)) \rightarrow (11 \ 22 \ 33)$
- $(\text{map list } \ '(1 \ 2) \ '(a \ z) \ '(q \ s)) \rightarrow ((1 \ a \ q) \ (2 \ z \ s))$

LA FONCTION APPLY

- `(apply f l)` applique la fonction `f` à l'ensemble des éléments de la liste `l`
- `(apply + '(1 2 3 4))` → 10
- `(apply (lambda (z) (cons z '(k))) '((1 2 a b)))`
→ `((1 2 a b) k)`
- La fonction `apply` n'est pas très utile en elle-même
- Elle sert pour exploiter le résultat d'un `map`, qui est toujours une liste

MAP-APPLY : UN EXEMPLE

- Une fonction pour calculer le produit scalaire de deux vecteurs :

$$L_1 = '(x_1 \ x_2 \ \dots \ x_n)$$

$$L_2 = '(y_1 \ y_2 \ \dots \ y_n)$$

$$\text{produit scalaire} = x_1y_1 + x_2y_2 + \dots + x_ny_n$$

```
(define scalaire ; → nombre
  (lambda (L1 L2) ; listes de nb
    (apply + (map * L1 L2))))
```

DÉFINITIONS « FORMELLES »

- Soit f_1 une fonction unaire
 $(\text{map } f_1 '(x_1 \dots x_n)) \rightarrow (f_1(x_1) \dots f_1(x_n))$
- Soit f_2 une fonction binaire
 $(\text{map } f_2 '(x_1 \dots x_n) '(y_1 \dots y_n))$
 $\rightarrow (f_2(x_1, y_1) \dots f_2(x_n, y_n))$
- Soit f_3 une fonction ternaire
 $(\text{map } f_3 '(x_1 \dots x_n) '(y_1 \dots y_n) '(z_1 \dots z_n))$
 $\rightarrow (f_3(x_1, y_1, z_1) \dots f_3(x_n, y_n, z_n))$
- ... et ainsi de suite
- Soit f_n une fonction n-aire
 $(\text{apply } f_n '(x_1 \dots x_n)) \rightarrow f_n(x_1, \dots, x_n)$

MAP-APPLY : LES DIFFÉRENCES (1)

MAP

- Prend autant de listes que l'arité de la fonction, toutes les listes étant de même longueur

APPLY

- Prend une liste comme argument, la longueur de cette liste étant égale à l'arité de la fonction

MAP-APPLY : LES DIFFÉRENCES (2)

MAP

- Applique la fonction à **chaque élément** de la liste (ou des listes)
- Retourne toujours **une liste de résultats** du type de celui de la fonction

APPLY

- Applique la fonction à **l'ensemble des éléments** de la liste
- Retourne **un résultat** du type de celui de la fonction

EXEMPLE DE DIFFÉRENCE

- (map list '(2 3 5)) → ((2) (3) (5))
- (apply list '(2 3 5)) → (2 3 5)