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

```
Fonction CalculSurListe
(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 donnés 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 argument 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 arguments,
                       ; 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 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
- (map (lambda (a b) (+ a b)) '(1 2 3) '(10 20 30))
  → (11 22 33)
- (map + '(1 2 3) '(10 20 30)) → (11 22 33)
- (map list '(1 2) '(a z) '(q s)) → ((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 :
  $L_1 = '(x_1\ x_2\ \ldots\ x_n)$
  $L_2 = '(y_1\ y_2\ \ldots\ y_n)$
  produit scalaire = $x_1y_1 + x_2y_2 + \ldots + x_ny_n$

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

Définitions « formelles »

- Soit f_1 une fonction unaire
  (map f_1 '(x_1 … x_n)) → ($f_1(x_1)$ … $f_1(x_n)$)
- Soit f_2 une fonction binaire
  (map f_2 '(x_1 … x_n) '(y_1 … y_n)) → ($f_2(x_1,y_1)$ … $f_2(x_n,y_n)$)
- Soit f_3 une fonction ternaire
  (map f_3 '(x_1 … x_n) '(y_1 … y_n) '(z_1 … z_n))
  → ($f_3(x_1,y_1,z_1)$ … $f_3(x_n,y_n,z_n)$)
- … et ainsi de suite
- Soit f_n une fonction n-aire
  (apply f_n '(x_1 … x_n)) → $f_n(x_1, \ldots, x_n)$

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