

# Scheme - TP n° 2

Université Paris I. Maîtrise de Logique. Année 2004-2005.  
Nicolas Stroppa - [stroppa@enst.fr](mailto:stroppa@enst.fr)

## 1 Notions complémentaires sur les fonctions

Nous avons vu comment définir des variables et des fonctions d'arité fixée. Il est aussi possible de définir des fonctions d'arité quelconques avec la construction (`define (f . x) ...`), où `x` représente la liste des arguments passés à la fonctions. À l'aide de cette nouvelle construction, redéfinissez les fonctions `compose-fonctions`, `curryfie`, `dec Curryfie` qui peuvent désormais prendre un nombre quelconques d'arguments.

## 2 Encore des listes

Définir les fonctions suivantes:

- `max-list` qui renvoie le maximum d'une liste
- `max-elem` qui renvoie le plus grand des éléments passés en paramètre (nombre quelconque).
- la fonction `map-list` qui applique une fonction à tous les arguments de la liste et qui renvoie le résultat.

## 3 Définition d'ensembles

On a vu que la structure de liste était omniprésente en Scheme. En réalité, on peut utiliser les listes comme moyen de définir des structures plus abstraites.

On définira l'ensemble vide par la liste vide et un ensemble non-vide par une liste linéaire ordonnée.

Définir:

- La variable `ensemble-vide`
- La fonction `ensemble-vide?`
- La fonction `ajouter-element`
- La fonction `union`
- La fonction `intersection`
- La fonction `inclusion`

## 4 Un peu de mathématique

À partir de la fonction factorielle le  $n$ -ième terme de la série de Taylor,  $x^i/i!$ .

Définir la fonction qui calcule la somme des  $n$  premiers termes. Question subsidiaire: quelle est la limite de cette suite ?

## 5 Les fonctions de tests

Regardez les programmes suivants et testez les différentes fonctions `=`, `eq?` et `equal?`. Concluez sur leurs rôles respectifs.

```
(define x '(salut toi))
(define y '(salut toi))
(equal? x y)
(eq? x y)
```

```
(define x 'salut)
(define y 'salut)
(equal? x y)
(eq? x y)
```

## 6 Modifier la valeur des variables

Dans le précédent TP, nous avons vu comment définir des variables à l'aide de `define` et `let`. Nous allons maintenant étudier comment modifier la valeur d'une variable déjà définie. Pour cela, testez le programme suivant.

```
(define y 0)
(set! y 9)
```

Concluez sur le rôle de `set!`. Notez que le symbole `!` est utilisé pour attirer l'attention sur le fait que la fonction modifie une variable. Pour représenter le test, le symbole était `?` (comme dans `null?`).

Utilisez `set!` dans une fonction de façon à modifier une variable locale. (`set!` permet de modifier des variables définies à l'aide de `define` ou `let`.)

Étudiez le programme suivant et créer plusieurs compteurs. Commentez.

```
(define (creer-compteur)
  (let ((i 0))
    (list (lambda () i)
          (lambda () (set! i (+ i 1)) i)
          (lambda () (set! i (- i 1)) i)
          )))
```

```
(define (valeur c) ((car c)))  
(define (incremente-compteur c) ((cadr c)))  
(define (decremente-compteur c) ((caddr c)))
```

À l'aide de variables locales et de `set!`, améliorer la version de la fonction de Fibonacci.

## 7 Éléments de corrigé

```
;; Partie 1

(define (carre x) (* x x))
(define (compose-functions l)
  (if (null? (cdr l))
      (eval (car l))
      (lambda (x)
        ((eval (car l)) ((compose-functions (cdr l)) x)))))

;; Partie 2
(define (max-list l)
  (if (null? (cdr l)) (car l)
      (let ((max-tail (max-list (cdr l)))
            (max-first (car l)))
        )
      (if (> max-first max-tail) max-first max-tail)))

(define (max-elem . l)
  (max-list l))

(define (map-list f l)
  (if (null? l)
      '()
      (cons (f (car l))
            (map-list f (cdr l)))))

;; Partie 3
(define ensemble-vide '())
(define (ensemble-vide? e)
  (null? e))

(define (ajouter-element x e)
  (cond ((ensemble-vide? e) (list x))
        ((< x (car e)) (cons x e))
        ((= x (car e)) e)
        (#t (cons (car e) (ajouter-element x (cdr e)))))

(define (union e1 e2)
  (if (ensemble-vide? e1)
      e2
```

```

    (ajouter-element (car e1) (union (cdr e1) e2))))

(define (appartient? x e2)
  (cond ((ensemble-vide? e2) #f)
        ((< x (car e2)) #f)
        ((= x (car e2)) #t)
        (#t (appartient? x (cdr e2)))))

(define (intersection e1 e2)
  (cond ((ensemble-vide? e1) ensemble-vide)
        ((appartient? (car e1) e2) (ajouter-element (car e1) (intersection (cdr e1) e2)))
        (#t (intersection (cdr e1) e2))))

(define (inclusion? e1 e2)
  (if (ensemble-vide? e1)
      #t
      (and (appartient? (car e1) e2) (inclusion? (cdr e1) e2))))

(define (creer-ensemble . l)
  (define (creer-ensemble-local l)
    (if (null? l)
        ensemble-vide
        (ajouter-element (car l) (creer-ensemble-local (cdr l)))))
  (creer-ensemble-local l))

;; Partie 4
(define (fact n)
  (if (<= n 1) 1 (* n (fact (- n 1)))))
(define (series f n)
  (if (zero? n) 0
      (+ (f n) (sum f (- n 1)))))
(define (e-power x n)
  (define (e-taylor i)
    (/ (expt x i) (fact i)))
  (+ #i1 (series e-taylor n)))

;Somme de la série = exp(x)

;; Partie 5
(define x '(salut toi))
(define y '(salut toi))
(equal? x y)

```

```
(eq? x y)

(define x 'salut)
(define y 'salut)
(equal? x y)
(eq? x y)

;; Partie 6
(define y 0)
(set! y 9)

(define (creer-compteur)
  (let ((i 0))
    (list (lambda () i)
          (lambda () (set! i (+ i 1)) i)
          (lambda () (set! i (- i 1)) i)
          )))
(define (valeur c) ((car c)))
(define (incremente-compteur c) ((cadr c)))
(define (decremente-compteur c) ((caddr c)))
```