(Séquence 5.1

map

Appliquer une fonction à une liste

- Présentation de plusieurs fonctions différentes
  - `liste-carres`
  - `liste-racines-carrees`
  - `liste-positive?`
- Le schéma récursif commun
- Passer la fonction en paramètre : la fonctionnelle `map`

Fonction `liste-carres`

```
;;; liste-carres: LISTE[Nombre] -> LISTE[Nombre]
;;; (liste-carres L) rend la liste des carrés
;;; des éléments de L.
(define (liste-carres L)
  (if (pair? L)
      (cons (carre (car L))
            (liste-carres (cdr L)) )
      (list) ) )
```

```
(liste-carres (list 1 2 3 4 5 6))
   → (1 4 9 16 25 36)
```

Fonction `liste-racines-carrees`

```
;;; liste-racines-carrees:
;;;      LISTE[Nombre] -> LISTE[Nombre]
;;; (liste-racines-carrees L) rend la liste des
;;; racines carrées des éléments de L.
;;; HYPOTHÈSE: les nombres de L sont positifs
(define (liste-racines-carrees  L)
  (if (pair? L)
      (cons (sqrt (car L))
            (liste-racines-carrees (cdr L)) )
      (list)))
```

```
(liste-racines-carrees (list 1 4 9 16 25))
   → (1 2 3 4 5)
```

Fonction `liste-positive?`

```
;;; liste-positive?: LISTE[Nombre] -> LISTE[bool]
;;; (liste- positive? L) rend la liste des booléens
;;; qui indiquent pour chaque élément de L s'il est
;;; positif.
(define (liste-positive?  L)
  (if (pair? L)
      (cons (positive? (car L))
            (liste-positive? (cdr L)) )
      (list)))
```

```
(liste-positive? (list 5 -9 0 4 8 -7))
  → (#t #f #f #t #t #f)
```

Vers un schéma de fonction

```
(define (liste-carres L)
  (if (pair? L)
      (cons (carre (car L))
            (liste-carres (cdr L)) )
      (list) ) )

(define (liste-racines-carrees  L)
  (if (pair? L)
      (cons (sqrt (car L))
            (liste-racines-carrees (cdr L)) )
      (list)))

(define (liste-positive?  L)
  (if (pair? L)
      (cons (positive? (car L))
            (liste-positive? (cdr L)) )
      (list)))
```

Un schéma de fonction récursive

Les trois définitions précédentes suivent le même **schéma récursif** :

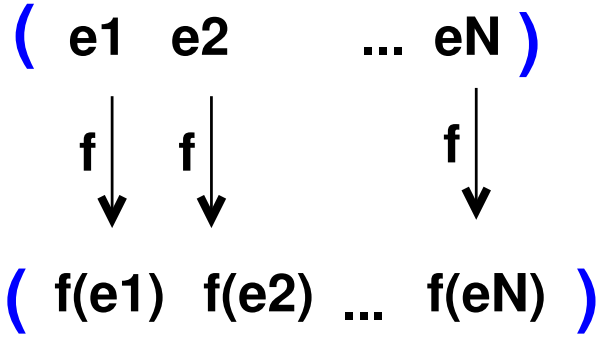
```
(define (fn-sur-liste L)
  (if (pair? L)
      (cons (fn-sur-elem (car L))
            (fn-sur-liste (cdr L)) )
      (list) ) )
```

La seule différence est la fonction *fn-sur-elem* à appliquer. Passer la fonction *fn-sur-elem* en argument, c'est ***abstraire*** !

Une fonctionnelle `map`

(cf. carte de référence)

```
;;; map: (alpha -> beta)
;;;      * LISTE[alpha] -> LISTE[beta]
;;; (map fn L) rend la liste dont les éléments
;;; résultent de l'application de la fonction fn
;;; aux éléments de L
(define (map fn L)
  (if (pair? L)
      (cons (fn (car L)) (map fn (cdr L)))
      (list)))
```



Applications de la fonction `map`

```
(liste-carres (list 1 2 3 4))
  ≡ (map carre (list 1 2 3 4))

(liste-racines-carrees (list 16 25 36))
  ≡ (map sqrt (list 16 25 36))
```

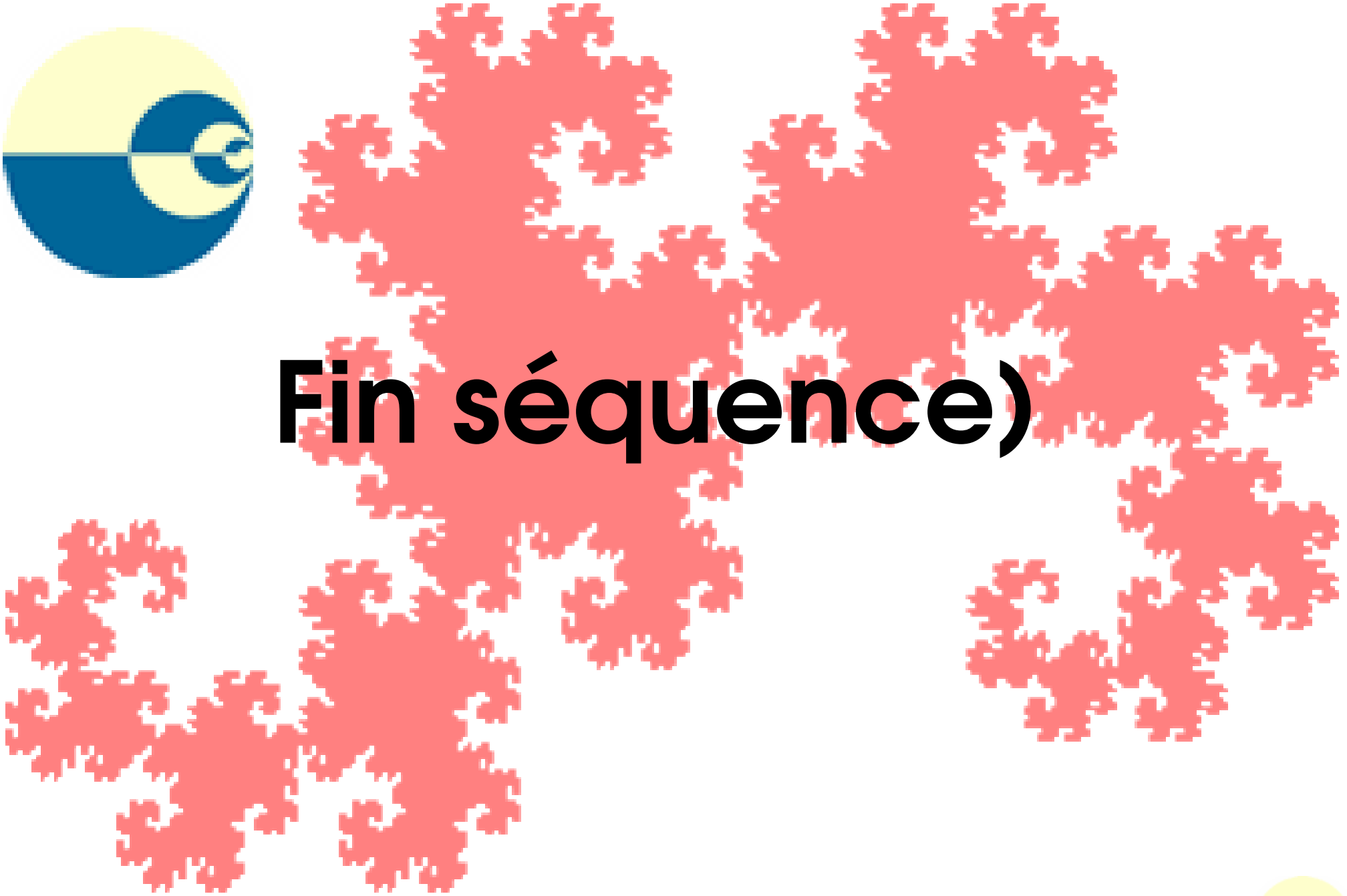
```
(map f (list e1 e2 ... eN))
  ≡ (list (f e1) (f e2) ... (f eN))
```

fonction interne et `map`

```
;;; verif-entre-bornes :
;;;  Nombre * Nombre * LISTE[Nombre] -> LISTE[bool]
;;; (verif-entre-bornes borneInf borneSup L) rend
;;; une liste de booléens vérifiant si les termes
;;; de L sont entre borneInf et borneSup

(define (verif-entre-bornes borneInf borneSup L)
  ;; entre-bornes? : Nombre -> bool
  ;; (entre-bornes? x) vérifie si x est entre
  ;; borneInf et borneSup
  (define (entre-bornes-inf-sup? x)
    (and (<= borneInf x) (<= x borneSup)))

  (map entre-bornes-inf-sup? L))
```



Fin séquence)