(Séquence 5.2

filtre

Filtrer une liste par un prédicat

Un autre problème : filtrer les éléments d'une liste par un prédicat *pred ?*

- soit *L*1 une liste d'éléments de type *alpha*,
- et *pred ?* un prédicat : *alpha* $\rightarrow$ *bool* ;
- on applique *pred ?* sur chaque élément de la liste *L*1
- pour obtenir une liste *L2* qui contient UNIQUEMENT les éléments vérifiant *pred ?*

```
;;; filtre:
;;;  (alpha -> bool) * LISTE[alpha] -> LISTE[alpha]
;;; (filtre pred? L) rend la liste des éléments de
;;; L qui vérifient le prédicat pred?
```

La fonction `filtre-pairs`

```
;;; filtre-pairs: LISTE[int] -> LISTE[int]
;;; (filtre-pairs L) retourne la liste des éléments
;;; pairs de L.
(define (filtre-pairs L)
  (if (pair? L)
      (if (even? (car L))
          (cons (car L) (filtre-pairs (cdr L)))
          (filtre-pairs (cdr L)))
      (list)))
```

```
(filtre-pairs (list 1 2 3 5 8 6))
   → (2 8 6)
```

La fonction `filtre-impairs`

```
;;; filtre-impairs: LISTE[int] -> LISTE[int]
;;; (filtre-impairs L) retourne la liste des éléments
;;; impairs de L.
(define (filtre-impairs L)
  (if (pair? L)
      (if (odd? (car L))
          (cons (car L) (filtre-impairs (cdr L)))
          (filtre-impairs (cdr L)))
      (list)))
```

```
(filtre-impairs (list 1 2 3 5 8 6))
  → (1 3 5)
```

Vers un schéma de fonction

```
(define (filtre-pairs L)
  (if (pair? L)
      (if (even? (car L))
          (cons (car L) (filtre-pairs (cdr L)))
          (filtre-pairs (cdr L)))
      (list)))

(define (filtre-impairs L)
  (if (pair? L)
      (if (odd? (car L))
          (cons (car L) (filtre-impairs (cdr L)))
          (filtre-impairs (cdr L)))
      (list)))
```

Schéma de fonction `filtre`

Les deux définitions précédentes suivent le même schéma récursif :

```
(define (schema-filtre-pred L)
  (if (pair? L)
      (if (pred-sur-elem? (car L))
          (cons (car L)
                (schema-filtre-pred (cdr L)))
          (schema-filtre-pred (cdr L)))
      (list)))
```

Il reste à abstraire vis-à-vis du prédicat *pred-sur-elem ?*

La fonctionnelle `filtre`

```
;;; filtre:
;;;  (alpha -> bool) * LISTE[alpha] -> LISTE[alpha]
;;; (filtre pred? L) rend la liste des éléments de
;;; L qui vérifient le prédicat pred?
(define (filtre pred? L)
  (if (pair? L)
      (if (pred? (car L))
          (cons (car L) (filtre pred? (cdr L)))
          (filtre pred? (cdr L)))
      (list)))
```

```
(filtre even? (list 1 2 3 4 5))
  → (2 4)
(filtre odd? (list 1 2 3 4 5))
  → (1 3 5)
(filtre integer? (list 2 2.5 3 3.5 4))
  → (2 3 4)
```

Points communs entre `map` et `filtre`

```
;;; map:
;;;   (alpha -> beta) * LISTE[alpha] -> LISTE[beta]
;;; filtre:
;;;   (alpha -> bool) * LISTE[alpha] -> LISTE[alpha]
```

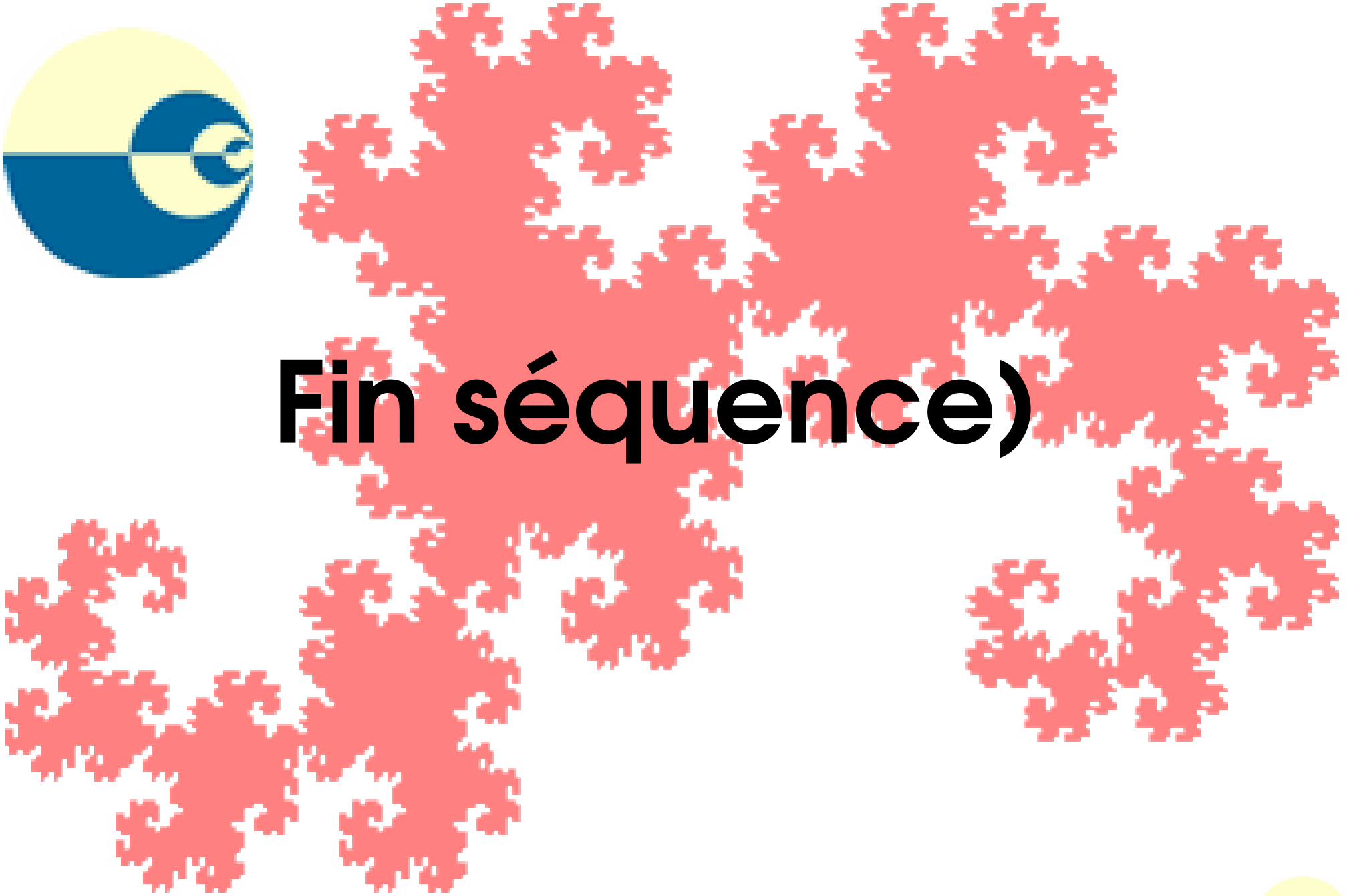
- Ce sont des fonctionnelles (elles reçoivent en argument une fonction)
- La fonction passée en premier argument a comme type de données le type des éléments de la liste de second argument
- Elles rendent une liste

Différences entre `map` et `filtre`

```
;;; map:
;;;   (alpha -> beta) * LISTE[alpha] -> LISTE[beta]
;;; filtre:
;;;   (alpha -> bool) * LISTE[alpha] -> LISTE[alpha]
```

- Le type du résultat de la fonction passée en argument : quelconque pour `map` et booléen pour `filtre`
- La longueur de la liste donnée et de la liste résultat (même longueur pour `map`)
- Les éléments de la liste résultat de filtre sont des éléments de sa liste donnée.

```
(map integer? (list 2 2.5 3 3.5 4))
  → (#t #f #t #f #t)
(filtre integer? (list 2 2.5 3 3.5 4))
  → (2 3 4)
```



Fin séquence)