(Séquence 4.6

Exemples commentés

Quelques exemples illustrant

- Récursion sur des listes (append, ajout-en-fin, somme-cumulee)
- Complexité

Buts

Les spécifications des fonctions à écrire :

```
;;; ajout-en-fin: alpha * LISTE[alpha]
;;;                    -> LISTE[alpha]
;;; (ajout-en-fin x L ) rend la liste obtenue en
;;; ajoutant x à la fin de la liste L
```

```
;;; append: LISTE[alpha] * LISTE[alpha]
;;;                    -> LISTE[alpha]
;;; (append L1 L2) rend la concaténation de L1 et
;;; de L2
```

Ajout suffixe

```
;;; ajout-en-fin: alpha * LISTE[alpha]
;;;         -> LISTE[alpha]
;;; (ajout-en-fin x L ) rend la liste obtenue en
;;; ajoutant x à la fin de la liste L

(define (ajout-en-fin x L)
  (if (pair? L)
      (cons (car L)
            (ajout-en-fin x (cdr L)) )
      (list x) ) )
```

Quelquefois nommée `snoc` !

Complexité de l'ajout suffixe

On mesure ici le nombre d'appels à `cons` (qui consomme de la mémoire)

```
(ajout-en-fin 4 (list 1 2 3) )

|(ajout-en-fin 4 (1 2 3))
| (ajout-en-fin 4 (2 3))
| |(ajout-en-fin 4 (3))
| | (ajout-en-fin 4 ())
| | (4)
| |(3 4)
| (2 3 4)
|(1 2 3 4)
```

Si `L` a n termes, `snoc` (c'est-à-dire `ajout-en-fin`) effectue $n+1$ appels à `cons`. Le coût est donc ***linéaire*** alors que le coût de `cons` est constant (mais pas nul !).

Concaténation de listes

```
;;; append: LISTE[alpha] * LISTE[alpha]
;;;                     -> LISTE[alpha]
;;; (append L1 L2) rend la concaténation de L1 et
;;; de L2

(define (append L1 L2)
  (if (pair? L1)
      (cons (car L1)
            (append (cdr L1) L2))
      L2))
```

Complexité de la concaténation de listes

```
(append (list 1 2 3) (list 5 6 7 8))

|(append (1 2 3) (5 6 7 8))
| (append (2 3) (5 6 7 8))
| |(append (3) (5 6 7 8))
| | (append () (5 6 7 8))
| | (5 6 7 8)
| |(3 5 6 7 8)
| (2 3 5 6 7 8)
|(1 2 3 5 6 7 8)
```

Si `L1` a *n* termes, `append` effectue *n* appels à `cons`. Le coût est donc **linéaire** sur le premier argument (indépendamment de la longueur du second argument).

Concaténation de listes (variante)

et si la récursion portait plutôt sur `L2` ?

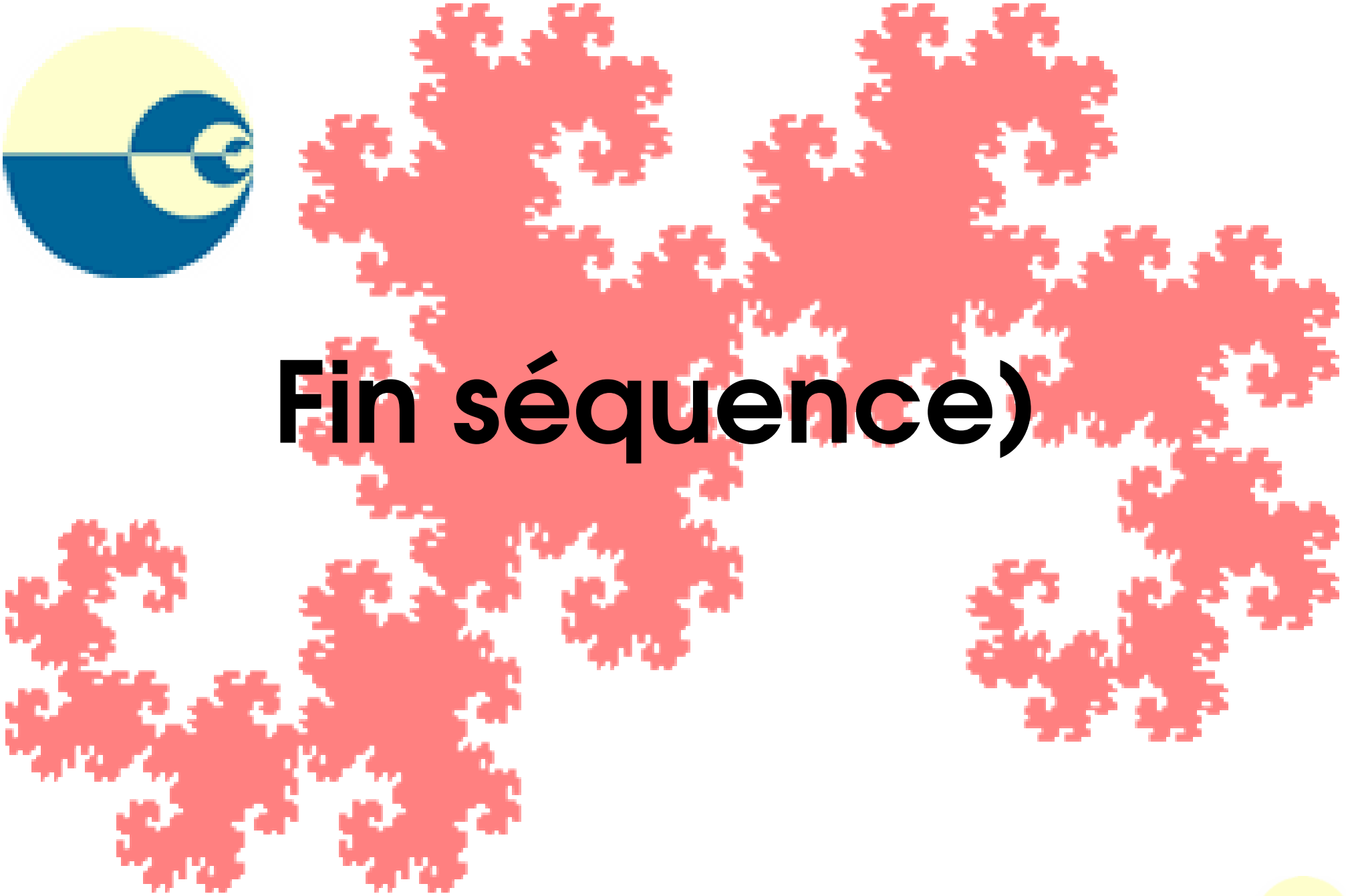
```
(define (appendr L1 L2)
  (if (pair? L2)
      (appendr (ajout-en-fin (car L2) L1)
               (cdr L2) )
      L1 ) )
```

Complexité de la concaténation de listes (variante)

La trace est trop longue pour être montrée. Vous pouvez l'obtenir vous-même avec MrScheme.

Si `L1` et `L2` ont n termes alors `appendr` effectue n appels à `ajout-en-fin` sur des listes de taille n puis $n+1$ puis $n+2$, etc. Le coût est donc, en nombre de `cons`, $\Sigma_{i=1}^{n}(n+i)$ c'est-à-dire de l'ordre de n^2 donc ***quadratique***.

Il faut donc faire attention au sens selon lequel on traite les listes !



Fin séquence)