(Séquence 4.1

Récursion sur listes

Définition d'une liste

Une ***liste*** est une ***structure de données*** qui regroupe une séquence d'éléments de même type.
En termes plus informatiques, une liste est une ***collection homogène ordonnée***.
Le type d'une telle liste se note : **LISTE(α)**

- Quelques listes :

```
(10 12 16 14)                LISTE[nat]
("ma" "me" "mes")            LISTE[string]
((10 12) (16 14 -1))         LISTE[LISTE[int]]
```

- La liste vide représentée par : `()` et se prononce « nil »

Structure de données

Trois familles de fonctions pour manipuler toute structure de données :

- Les ***constructeurs*** permettent de construire une donnée structurée
- Les ***accesseurs*** permettent d'accéder aux composants d'une donnée structurée
- Les ***reconnaisseurs*** permettent de reconnaître la nature d'une donnée structurée

Primitives relatives aux listes

- **Constructeurs** pour construire une liste : `list`, `cons`
- **Accesseurs** pour accéder aux parties d'une liste : `car`, `cdr` (prononcer « coudeur »)
- **Reconnaisseur** (prédicat) pour savoir si une valeur est une liste non vide : `pair?`

Deux constructeurs `list` et `cons`

La fonction `list` (cf. carte de référence)

```
;;; list: alpha * ... -> LISTE[alpha]
;;; (list v...) cree une liste dont les termes
;;; sont les arguments. (list) rend la liste vide
```

`list` est une fonction ***n-aire***, elle prend un nombre quelconque, non fixé, d'arguments

LISTE(int)

```
(list 1 2 (+ 2 1) (/ 8 2))
      → (1 2 3 4)
```

LISTE(string)

```
(list "je" "tu" (string-append "el" "le") "il")
      → ("je" "tu" "elle" "il")
```

LISTE(bool)

```
(list (= 2 3) (not #f) (> 2 3))
      → (#f #t #f)
```

Constructeur `cons`

Une liste est (récursivement) définie comme :

- la liste vide
- ou une liste non vide c'est-à-dire constituée :
  - d'un premier terme
  - et d'autres termes qui forment aussi une liste (vide ou pas).

```
;;; cons: alpha * LISTE[alpha] -> LISTE[alpha]
;;; (cons v L) rend la liste dont le premier élément
;;; est v et dont les éléments suivants sont les
;;; éléments de la liste L.
```

```
(cons (+ 5 5) (list 20 30 40))
      → (10 20 30 40)
(cons 10 (cons 20 (cons 30 (cons 40 (list)))))
      → (10 20 30 40)
```

Les accesseurs

```
;;; car: LISTE[alpha] -> alpha
;;; (car L) rend le premier élément de la liste L
;;; ERREUR lorsque L n'est pas une liste non vide

;;; cdr: LISTE[alpha] -> LISTE[alpha]
;;; (cdr l) rend la liste des termes de L sauf son
;;; premier élément.
;;; ERREUR lorsque L n'est pas une liste non vide

(car (list 10 20 30 40))
       → 10
(cdr (list 10 20 30 40))
       → (20 30 40)
```

Propriétés algébriques

- Pour toute liste L et toute valeur v

```
(car (cons v L)) ≡ v
(cdr (cons v L)) ≡ L
```

- Pour toute liste **non vide** L

```
(cons (car L) (cdr L)) ≡ L
```

Le reconnaisseur `pair?`

Pour savoir si une valeur est une liste non vide : le prédicat `pair?`

```
;;; pair?: valeur -> bool
;;; (pair? v) rend vrai ssi v a un car et un cdr,
;;; c'est-à-dire ssi v est une liste non vide.
```

```
(pair? (list 10 20 30 40))
       → #t
(pair? (list))
       → #f
```

Remarque super-importante

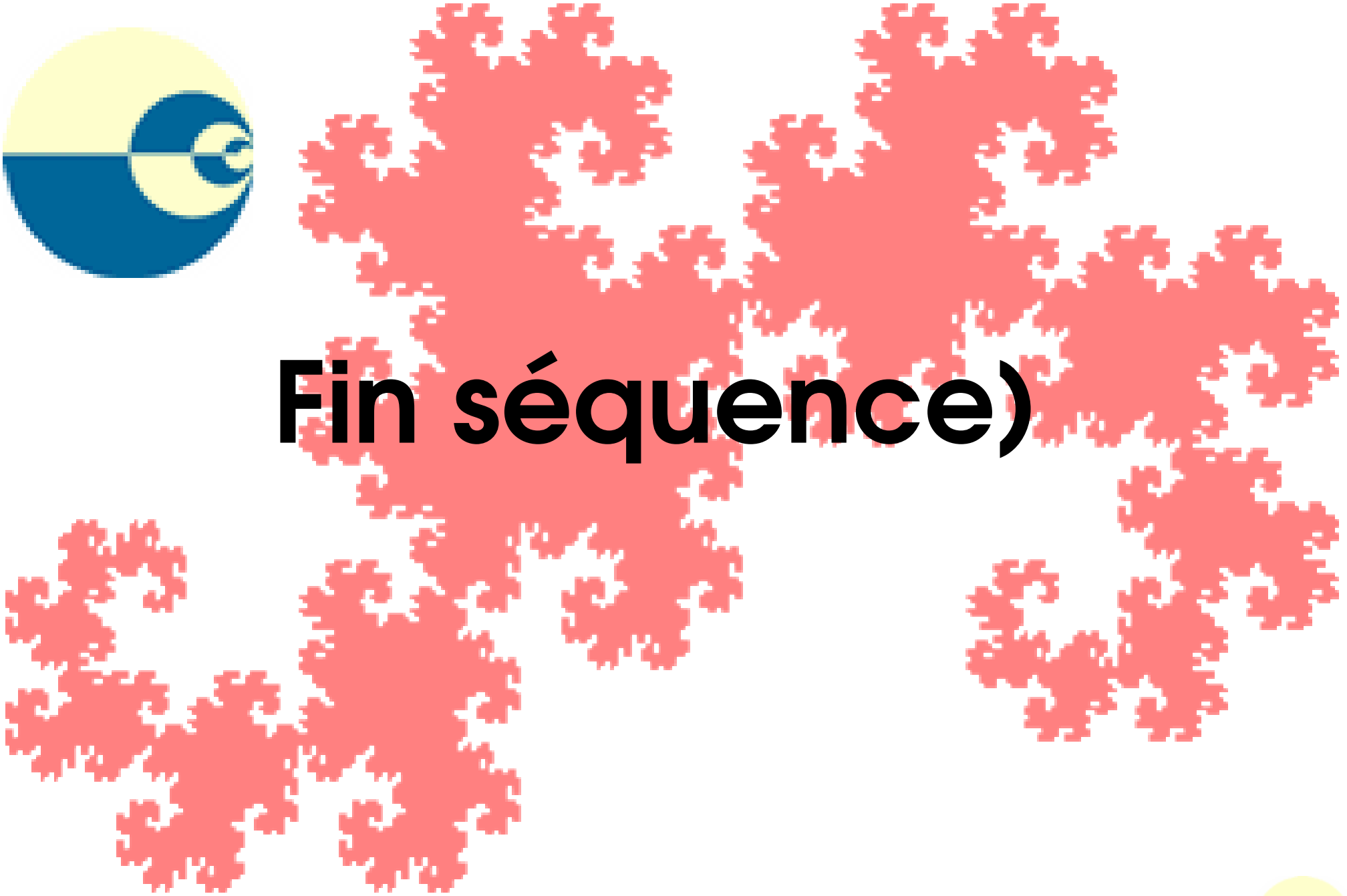
Jamais `car` (ou `cdr`) ne prendras sans que de `pair?` ne t'assureras !

Si l'on sait que `L` vérifie `pair?` alors on peut directement écrire

```
... (car L) ...
```

sinon on doit écrire :

```
(if (pair? L)
    ... (car L) ...
    ... )
```



Fin séquence)