# LI101 : Programmation Récursive

©Equipe enseignante Li101

Université Pierre et Marie Curie

Semestre : Automne 2013

## Cours 4 : Listes

# Plan du cours

1. Définition d'une liste
2. Primitives sur les listes
   - Constructeurs
   - Accesseurs
   - Reconnaisseurs
3. Définition de fonctions récursives sur les 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[$\alpha$]**

- Quelques listes :

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

- La liste vide représentée par : `()`

# Structure de données

Trois familles de fonctions pour manipuler une 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`
- **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...) crée une liste dont les termes sont
;;; les arguments
;;; (list) rend la liste vide
```

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

```
LISTE[nat] :
(list 1 2 (+ 2 1) (/ 8 2)) ->  (1 2 3 4)
LISTE[string] :
(list "je" "tu" "elle" "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.

```
;;; 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)
```

# Une construction récursive de liste

```
(define (mystere n)
  (if (> n 0)
      (cons n (mystere (- n 1)))
      (list) ) )
```

Quel est le résultat de `(mystere 5)` ?

```
;;; mystere: nat  -> LISTE[nat]
;;; (mystere n) rend la liste des entiers de n à 1,
;;; rend la liste vide si n = 0
```

# 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
```

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

# Le reconnaisseur pair?

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

```
;;; pair? : Valeur -> bool
;;; (pair? x) rend vrai ssi x  est une liste non vide.
```

```
(pair? (list 10 20 30 40)) → #T
(pair? (list)) → #F
(pair? 0) → #F
```

# Remarque super-importante

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

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

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

sinon on doit écrire :

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

# Somme des termes d'une liste

Sa spécification :

```
;;; somme: LISTE[Nombre] -> Nombre
;;; (somme L) rend la somme des éléments de L,
;;; rend 0 pour la liste vide
```

# Somme des termes d'une liste

- Lorsque la liste donnée n'est pas vide :
  la somme de ses éléments est égale au premier élément `(car L)` **plus** la somme des éléments du `cdr` de la liste

  ```
  (somme (list e1 e2 ... en))
   ≡ (+ e1 (somme (list e2 ... en)))
  ```

- Lorsque la liste donnée est vide :
  la somme de ses éléments est égale à 0, par convention

  ```
  (somme (list)) ≡ 0
  ```

# Une définition Scheme de somme

```
;;; somme: LISTE[Nombre] -> Nombre
;;; (somme L) rend la somme des éléments de L,
;;; rend 0 pour la liste vide
(define (somme L)
  (if (pair? L)
      (+ (car L) (somme (cdr L)))
      0 ) )
```

# Trace d'une évaluation

On évalue `(somme (list 1 4 6 20))`

```
|(somme (1 4 6 20))
| (somme (4 6 20))
| |(somme (6 20))
| | (somme (20))
| | |(somme ())
| | |0
| | 20
| |26
| 30
|31
```

# Longueur d'une liste

```
;;; longueur: LISTE[alpha] -> nat
;;; (longueur L) rend la longueur de la liste L
(define (longueur L)
  (if (pair? L)
      (+ 1 (longueur (cdr L)))
      0 ) )
```

Quelle sera la trace de `(longueur (list 5 10 8))` ?

# Typage d'une fonction

Vérifier la **cohérence** entre

- les types qui sont précisés dans la spécification
- ce qui sera calculé par le code de la définition

```
;;; f: alpha * beta -> gamma
;;; (f x y) rend ...
```

```
(define (f x y)
  (... x ... y ...) )
```

Cf. équations aux dimensions en physique

```
(define (mystere001 X)
  (if (pair? X)
      (+ (carre (car X)) (mystere001 (cdr X)))
      0 ) )

(define (mystere002 X)
  (if (pair? X)
      (cons (carre (car X)) (mystere002 (cdr X)))
      (list) ) )

(define (mystere003 X)
  (if (pair? X)
      (cons (positive? (car X)) (mystere003 (cdr X)))
      (list) ) )
```

# Signature (type) d'une fonction

```
(define (carre nbre)
  (* nbre nbre) )
;;; carre: ???    -> ???
;;; carre: Nombre -> Nombre

(define (mystere001 X)
  (if (pair? X)
      (+ (carre (car X)) (mystere001 (cdr X)))
      0 ) )
;;; mystere001:                  ???
;;; mystere001: ???               -> Nombre
;;; mystere001: LISTE[???]        -> Nombre
;;; mystere001: LISTE[Nombre] -> Nombre
```

# Somme des carrés d'une liste

```
;;; carre: Nombre -> Nombre
;;; (carre n) rend le carré du nombre n
(define (carre n)
  (* n n) )

;;; somme-carres: LISTE[Nombre] -> Nombre
;;; (somme-carres L) rend la somme des carrés des
;;; éléments de L ; rend 0 pour la liste vide
(define (somme-carres L)
  (if (pair? L)
      (+ (carre (car L)) (somme-carres (cdr L)))
      0 ) )
```

```
(somme-carres (list 5 10 8 4))
|(somme-carres (5 10 8 4))
| (somme-carres (10 8 4))
| |(somme-carres (8 4))
| | (somme-carres (4))
| | |(somme-carres ())
| | |0
| | 16
| |80
| 180
|205
```

# Signature (type) d'une fonction

```
(define (mystere002 X)
  (if (pair? X)
      (cons (carre (car X)) (mystere002 (cdr X)))
      (list) ) )
;;; mystere002:                    ???
;;; mystere002: ???                 -> LISTE[???]
;;; mystere002: LISTE[???]          -> LISTE[???]
;;; mystere002: LISTE[Nombre]       -> LISTE[???]
;;; mystere002: LISTE[Nombre]       -> LISTE[Nombre]
```

# Liste des carrés d'une liste

```
;;; 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 5 10 8 4))
|(liste-carres (5 10 8 4))
| (liste-carres (10 8 4))
| |(liste-carres (8 4))
| | (liste-carres (4))
| | |(liste-carres ())
| | |()
| | (16)
| |(64 16)
| (100 64 16)
|(25 100 64 16)
```

# Signature (type) d'une fonction

```
(define (mystere003 X)
  (if (pair? X)
      (cons (positive? (car X)) (mystere003 (cdr X)))
      (list) ) )
;;; mystere003:                  ???
;;; mystere003: ???               -> LISTE[???]
;;; mystere003: LISTE[???]        -> LISTE[???]
;;; mystere003: LISTE[Nombre]     -> LISTE[???]
;;; mystere003: LISTE[Nombre]     -> LISTE[bool]
```

Réécrire la spécification et la définition de la fonction `mystere003` pour que cela soit plus significatif.

# Schéma de récursion simple

Une liste est :

- soit vide
- soit constituée d'un premier *élément* `(car L)` et d'une *liste* restante `(cdr L)`

```
;;; fRec: LISTE[alpha] -> ...
(define (fRec L)
  (if (pair? L)
      (combinaison (car L)
                   (fRec (cdr L)))
      cas-liste-vide ) )
```

# Travail avant le prochain TD/TP

- Points traités :
  - Notion de liste
  - Distinguer `list` et `cons`
  - Définitions récursives sur les listes
  - Signature d'une fonction
- Être capable d'écrire, sans l'aide des notes et du cours, toutes les spécifications et définitions des fonctions présentées