PLAN DU COURS 4

« Programmation récursive »

Les listes en Scheme

- Définition d'une liste
- Primitives sur les listes
  - Constructeurs
  - Accesseurs
  - Reconnaisseurs
- 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.

Le type d'une telle liste se note : **LISTE[α]**

- Quelques listes :

```
(10 12 16 14)
("ma" "me" "mes")
((10 12) (16 14))
```

- La liste vide représentée par : `()`

POUR UNE STRUCTURE DE DONNÉES

Les fonctions de base pour manipuler une structure de données sont de trois types :

- Les **constructeurs** permettent de construire une structure de données
- Les **accesseurs** permettent d'accéder aux éléments de la structure de données
- Les **reconnaisseurs** permettent de savoir quelle est la nature d'une donnée structurée

LES FONCTIONS PRIMITIVES SUR LES LISTES

Les fonctions de base sur les listes sont des fonctions primitives de Scheme

- **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
```

Un nombre quelconque, non fixé, d'arguments

DEM

```
(list 1 2 3 4)  → (1 2 3 4)
(list "je" "tu" "elle" "il")  → ("je" "tu" "elle" "il")
(list (= 2 3) (not #F) (> 2 3))  → (#F #T #F)
```


Constructeur cons

Une liste est (définition récursive) :

- soit la liste vide
- soit constituée :
  - du premier terme,
  - et du reste des termes qui forme 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)
```


Les accesseurs

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

;;; cdr: LISTE[alpha] ]/non vide/ -> 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

- 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`

Exemples *DEM*

```
(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? 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 prendra sans que de `pair?` tu 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) ...
    ... )
```


La récursivité sur les listes

Un exemple : la fonction `somme`

- Sa spécification :

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


- La définition récursive de la somme
  - Lorsque la liste donnée n'est pas vide : la somme de ses éléments est égale au premier élement `(car L)` + la somme des éléments du `cdr` de la liste
  - Lorsque la liste donnée est vide : la somme de ses éléments est égale à 0, par convention

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 ) )
```

DEM

Trace d'une exécution

On exécute `(somme (list 6 7 2))`

```
|(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 donnée
(define (longueur L)
  (if (pair? L)
      (+ 1 (longueur (cdr L)))
      0 ))
```

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

Cohérence dans l'écriture d'une fn

Vérifier la **cohérence** entre

- Les types qui sont précisés dans la spécification
- Ce qui sera calculé par le code de la définition

Pour cela, vérifier après chaque écriture d'un couple spécification/définition

- La cohérence entre le type des variables dans la spécification et leur utilisation dans la définition
- La cohérence entre le type attendu et la valeur calculée

Cf. équations aux dimensions en physique

DEM

```
(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 carres 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: ??? -> 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: ??? -> 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* et d'une *liste* restante `(car L)` est un *élément*, `(cdr L)` est une *liste*

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


Travail avant le prochain TD/TP

- Notion de liste
- Définitions récursives sur les listes
- Signature d'une fonction
- Faire les « exercices d'assouplissement »
- Être capable d'écrire, sans l'aide des notes et du cours, toutes les spécifications et définitions des fonctions présentées