PLAN DU COURS 6

« Programmation récursive »

Fonctionnelles sur les listes

- La fonctionnelle `map`
- La fonctionnelle `filtre`
- La fonctionnelle `reduce`

FONCTIONNELLES

Une **fonctionnelle** est une fonction qui prend une fonction en argument (ou renvoie une fonction en résultat)

Exemple : la fonctionnelle `map`, qui applique une fonction f à chaque élément d'une liste.

;;; *map: (alpha -> beta) * LISTE[alpha] -> LISTE[beta]*
;;; *(map f L) applique récursivement la fonction f à tous les éléments de L*
;;; *et renvoie la liste ainsi obtenue.*

- soit $L1$ une liste d'éléments de type $alpha$,
- et f une fonction : $alpha \rightarrow beta$;
- on applique f sur chaque élément de la liste $L1$,
- pour obtenir une liste $L2$ d'éléments de type $beta$.

APPLIQUER UNE FONCTION À UNE LISTE

- Présentation de plusieurs fonctions différentes
  `liste-carres`
  `liste-racines-carrees`
  `liste-positive?`
- Le schéma récursif commun
- Passer la fonction en paramètre : la fonctionnelle `map`

FONCTION `liste-carres`

```
;;; 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 1 2 3 4 5 6))` → `(1 4 9 16 25 36)`

FONCTION liste-racines-carrees

```
;;; liste-racines-carrees: LISTE[Nombre/>=0/] -> LISTE[Nombre]
;;; (liste-racines-carrees L) rend la liste des racines carrées des éléments
;;; de L.
(define (liste-racines-carrees  L)
  (if (pair? L)
      (cons (sqrt (car L)) (liste-racines-carrees (cdr L)))
      (list)))

 (liste-racines-carrees (list 1 4 9 16 25))  → (1 2 3 4 5)
```


FONCTION liste-positive?

```
;;; liste-positive?: LISTE[Nombre] -> LISTE[bool]
;;; (liste- positive? L) rend la liste des booléens qui indiquent pour chaque
;;; élément de L s'il est positif ou non.
(define (liste-positive?  L)
  (if (pair? L)
      (cons (positive? (car L)) (liste-positive? (cdr L)))
      (list)))

 (liste-positive? (list 5 -9 0 4 8 -7))  → (#T #F #F #T #T #F)
```


VERS UN SCHÉMA DE FONCTION

```
(define (liste-carres L)
  (if (pair? L)
      (cons (carre (car L))  (liste-carres (cdr L)))
      (list) ) )

(define (liste-racines-carrees  L)
  (if (pair? L)
      (cons (sqrt (car L)) (liste-racines-carrees (cdr L)))
      (list)))

(define (liste-positive?  L)
  (if (pair? L)
      (cons (positive? (car L)) (liste-positive? (cdr L)))
      (list)))
```


UN SCHÉMA DE FONCTION RÉCURSIVE

Les trois définitions précédentes suivent le même **schéma récursif** :

```
(define (FN-SUR-LISTE L)
  (if (pair? L)
      (cons (fn-sur-elem (car L))
            (FN-SUR-LISTE (cdr L)) )
      (list) ) )
```

La seule différence est la fonction *fn-sur-elem* à appliquer.
Passer la fonction *fn-sur-elem* en argument, c'est *abstraire* !

Une fonctionnelle map

(cf. carte de référence)

```
;;; map: (alpha -> beta) * LISTE[alpha] -> LISTE[beta]
;;; (map fn L) rend la liste dont les éléments résultent
;;; de l'application de la fonction fn aux éléments de L
(define (map fn L)
  (if (pair? L)
      (cons (fn (car L)) (map fn (cdr L)))
      (list)))
```


Applications de la fonction map

```
(liste-carres (list 1 2 3 4))
  ≡ (map carre (list 1 2 3 4))

(liste-racines-carrees (list 16 25 36))
  ≡ (map sqrt (list 16 25 36))
```

Autres exemples :

```
(map even? (list 1 2 3 4))

(map odd? (list 1 2 3 4 5))

(map list (list 1 2 3 4))
```


Trace de la fonction map

On exécute `(map even? (list 1 2 3 4))`

```
(map)
|(map #<primitive:even?> (1 2 3 4))
| (map #<primitive:even?> (2 3 4))
| |(map #<primitive:even?> (3 4))
| | (map #<primitive:even?> (4))
| | |(map #<primitive:even?> ())
| | |()
| | (#t)
| |(#f #t)
| (#t #f #t)
|(#f #t #f #t)
(#f #t #f #t)
```


Filtrer une liste par un prédicat

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

- soit $L1$ une liste d'éléments de type $alpha$,
- et *pred ?* un prédicat : $alpha \rightarrow bool$;
- on applique *pred ?* sur chaque élément de la liste $L1$
- 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 passer le prédicat *pred-sur-elem ?* en paramètre !

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.

Exemples

```
(let ((L (list 2 2.5 3 3.5 4))) (map integer? L))  → (#T #F #T #F #T)
(let ((L (list 2 2.5 3 3.5 4))) (filtre integer? L))  → (2 3 4)
```


Intermède algébrique

(if α (f β) (f γ))
$\equiv$ (f (if α β γ))

(if α (β arg) (γ arg))
$\equiv$ ((if α β γ) arg)

Combiner les éléments d'une liste

Un autre problème : Combiner entre eux les éléments d'une liste, à l'aide d'un opérateur binaire fn

- soit $L1$ une liste d'éléments de type $alpha$,
- et fn une fonction : $alpha * beta \rightarrow beta$;
- on applique fn sur chaque élément de la liste $L1$ (en démarrant avec un élément de base de type $beta$),
- pour obtenir un élément de type $beta$.

```
;;; reduce: (alpha * beta -> beta)  *  beta * LISTE[alpha] -> beta
;;; (reduce fn base L) rend fn(e1, fn(e2, ... fn(en, base) ...))
```


La fonction 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 ) )
```


La fonction overlay-liste

```
;;; overlay-liste: LISTE[Image] -> Image
;;; (overlay-liste L) produit une image superposant
;;; les contenus des images de la liste L
(define (overlay-liste L)
  (if (pair? L)
      (overlay (car L)
               (overlay-liste (cdr L)))
      (image-vide) ) )
```


Vers un schéma de fonction

```
(define (somme L)
  (if (pair? L)
      (+ (car L)
         (somme (cdr L)))
      0 ) )


(define (overlay-liste L)
  (if (pair? L)
      (overlay (car L)
               (overlay-liste (cdr L)))
      (image-vide) ) )
```

La fonctionnelle reduce

Passer en paramètres : l'opérateur binaire et l'élément de base

```
;;; reduce: (alpha * beta -> beta)  *  beta * LISTE[alpha] -> beta
;;; (reduce fn base L) rend fn(e1, fn(e2, ... fn(en, base) ...))
(define (reduce fn base L)
  (if (pair? L)
      (fn (car L)
          (reduce fn base (cdr L)))
      base ) )
```

Applications de `reduce` :

```
(reduce + 0 (list 1 2 3 4 5))  → 15
(reduce * 1 (list 1 2 3 4 5))  → 120
```


Ou encore, directement et sans récursion apparente :

```
(define (superposition L)
  (reduce overlay (image-vide) L) )
```


Trace de la fonction reduce

Exécution de `(reduce * 1 (list 1 2 3))` :

```
|(reduce #<primitive:*> 1 (1 2 3))
| (reduce #<primitive:*> 1 (2 3))
| |(reduce #<primitive:*> 1 (3))
| | (reduce #<primitive:*> 1 ())
| | 1
| |3
| 6
|6
```