# PLAN DU COURS 5

« Programmation récursive »

Des rappels au travers d'exemples

- Schéma de récursion simple sur des listes
- Récursion sur des listes
- Bloc et nommage
- Spécification et définition





# UN SCHÉMA DE RÉCURSION SUR LES LISTES

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)
      (une-combinaison (car L)
                       (fRec (cdr L)))
      cas-liste-vide ) )
```



# MÉTHODE D'ÉCRITURE

Pour écrire une fonction récursive, il est très fortement recommandé de suivre la méthode suivante :

1. décrire le calcul à partir du calcul sur des éléments « plus petits » (relation de récurrence),
2. déterminer les valeurs dites « valeurs de base » pour lesquelles :
   (a) la relation de récurrence n'est pas définie,
   (b) les valeurs d'appel de la fonction récursive, pas **strictement** « plus petites » que la donnée.
3. déterminer les valeurs de la fonction pour les valeurs de base.





# ARRÊT DE LA FONCTION RÉCURSIVE

Pour permettre un arrêt de la fonction récursive, il faut

1. une « diminution » dans l'argument de l'appel récursif
   - Si la récursivité se fait sur les entiers naturels, la valeur de l'argument de l'appel récursif doit diminuer (par exemple $n-1$, ou $n/2$, ou ...)
   - Si la récursivité se fait sur une liste, la longueur de la liste passée en argument de l'appel récursif doit être plus courte (par exemple `(cdr L)`).
2. un test d'arrêt sur les valeurs de base

## Concaténation de listes

```
;;; 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-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) ) )
```



```
(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)
```



```
;;; 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))
```

```
(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)
```





## Un exemple plus complexe

```
;;; somme-cumulee: LISTE[Nombre] -> LISTE[Nombre]
;;; (somme-cumulee L) rend la liste dont le premier élément est égal
;;; à la somme des éléments de L, dont le deuxième élément est égal à
;;; la somme des éléments de (cdr L) ... dont le dernier élément est égal
;;; au dernier élément de L.

;;; chaque  élément de la liste résultat est égal à la somme des éléments
;;; qui le suivent (sens large) dans la liste initiale

(somme-cumulee (list 4))  → (4)
(somme-cumulee (list 3 4))  → (7 4)
(somme-cumulee (list 2 3 4))  → (9 7 4)
(somme-cumulee (list 1 2 3 4))  → (10 9 7 4)
```



## Une première définition

`somme-cumulee`(L) est une liste dont

- le premier élément est égal à la somme du premier élément de L et du premier élément de la somme cumulée du `(cdr L)`
- le reste de la liste est la somme cumulée du `(cdr L)`

```
(define (somme-cumulee L)
  (if (pair? L)
      (if (pair? (cdr L))
          (let ((reste-fait (somme-cumulee (cdr L))))
            (cons (+ (car L) (car reste-fait))
                  reste-fait))
          L)
      (list)))
```



```
(somme-cumulee (list 1 2 3 4))

|(somme-cumulee (1 2 3 4))
| (somme-cumulee (2 3 4))
| |(somme-cumulee (3 4))
| | (somme-cumulee (4))
| | (4)
| |(7 4)
| (9 7 4)
|(10 9 7 4)
```

## Définition interne pour éviter des tests

```
(define (somme-cumulee L)
  ;; sc-non-vide: LISTE[Nombre]/non vide/ -> LISTE[Nombre]
  ;; (sc-non-vide L) == (somme-cumulee L)
  (define (sc-non-vide L)
    (if (pair? (cdr L))
        (let ((reste-fait (sc-non-vide (cdr L))))
          (cons (+ (car L) (car reste-fait))
                reste-fait))
        L))
  (if (pair? L)
      (sc-non-vide L)
      L))
```





## Une définition à proscrire

*À NE PAS IMITER !*

```
(define (somme-cumulee L)
  (if (pair? L)
      (if (pair? (cdr L))
          (cons (+ (car L) (car (somme-cumulee (cdr L))))
                (somme-cumulee (cdr L)))
          L)
      (list)))
```

**Pour éviter des recalculs, il faut nommer**



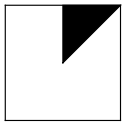
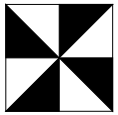
```
|(somme-cumulee (1 2 3 4))
| (somme-cumulee (2 3 4))
| |(somme-cumulee (3 4))
| | (somme-cumulee (4))
| | (4)
| | (somme-cumulee (4))
| | (4)
| |(7 4)
| |(somme-cumulee (3 4))
| | (somme-cumulee (4))
| | (4)
| | (somme-cumulee (4))
| | (4)
| |(7 4)
| (9 7 4)
| (somme-cumulee (2 3 4))
```



```
| |(somme-cumulee (3 4))
| | (somme-cumulee (4))
| | (4)
| | (somme-cumulee (4))
| | (4)
| |(7 4)
| |(somme-cumulee (3 4))
| | (somme-cumulee (4))
| | (4)
| | (somme-cumulee (4))
| | (4)
| |(7 4)
| (9 7 4)
|(10 9 7 4)
```

## UNE ILLUSTRATION DE `let` ET DE `let*`

```
;;; 4-rotations : Image -> Image
;;; (4-rotations image) rend l'image constituee de la superposition de l'image
;;; donnée et des images obtenues successivement par rotation d'un quart,
;;; d'un demi et de trois quarts de tour.
(4-rotations (filled-triangle 0 0 1 1 0 1))
```

la donnée-image :  →le résultat-image : 



```
;;; 4-rotations : Image -> Image
;;; (4-rotations image) rend l'image constituee de la superposition de l'image
;;; donnée et des images obtenues successivement par rotation d'un quart,
;;; d'un demi et de trois quarts de tour.
(define (4-rotations image)
  (overlay image
           (quarter-turn-right image)
           (quarter-turn-right
            (quarter-turn-right image))
           (quarter-turn-right
            (quarter-turn-right
             (quarter-turn-right image)))))
```



Une autre définition :

```
(define (4-rotations image)
  (let ((rota-1 (quarter-turn-right image)))

    (let ((rota-2 (quarter-turn-right rota-1)))

      (let ((rota-3 (quarter-turn-right rota-2)))

        (overlay image
                 rota-1
                 rota-2
                 rota-3)))))
```



Encore une autre définition :

```
(define (4-rotations3 image)
  (let* ((rota-1 (quarter-turn-right image))

         (rota-2 (quarter-turn-right rota-1))

         (rota-3 (quarter-turn-right rota-2)))

    (let ((image-rota1 (overlay image rota-1))
          (rota2-rota3 (overlay rota-2 rota-3)))

      (overlay image-rota1 rota2-rota3))))
```

## De l'énoncé d'un pb vers une fonction

Sur des exemples, nous allons montrer les différentes étapes qu'il est préférable de respecter pour passer de l'énoncé d'un problème vers une définition de fonction Scheme :

- de la formalisation du problème vers une spécification ;
- de la spécification vers une définition.

Deux exemples :

- À partir d'une image, avoir une liste de $n$ images identiques (révision de la récursion sur $n$)
- À partir d'une liste d'images, avoir une image constituée de la superposition de toutes les images de la liste (révision de la récursion sur une liste)



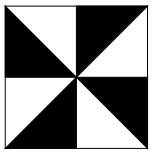


## Vers la spécification d'une fonction

- Choix du type de résultat de la fonction
- Choix du nom de la fonction (évoque la nature du résultat)
- Choix des arguments
  - Choix de leur ordre d'apparition
  - Choix de leur nom

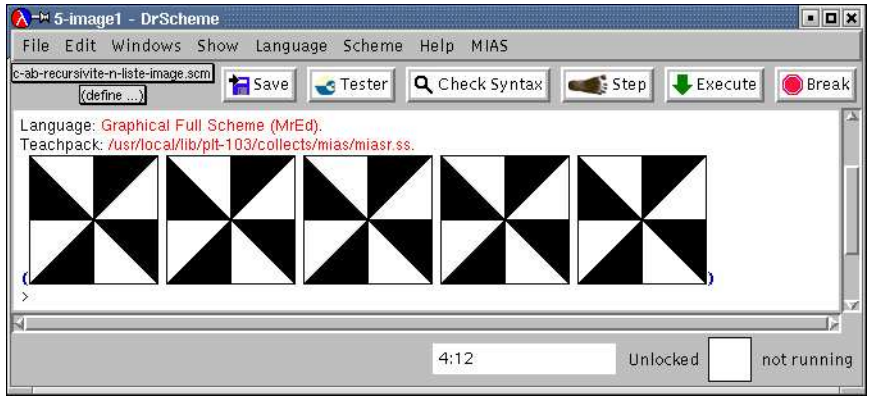


## Vers la spécification : une illustration

la donnée-image : 

le résultat-image :





## Spécification de `liste-images-idem`

```
;;; liste-images-idem : nat * image -> LISTE[image]
;;; (liste-images-idem n image) rend une liste de n images
;;; identiques à l'image donnée
```





## De la spécif. d'une fn vers sa définition

- Choisir l'algorithme
- Si la fonction est récursive, vérifier l'arrêt de la récursivité
- Mentionner les hypothèses d'utilisation
- Vérifier la **cohérence** entre
  - les types (variables/arguments et résultat) qui sont précisés dans la signature de la fonction
  - et ce qui est utilisé dans la définition, en particulier le type du résultat de l'appel récursif.
- Tester la fonction
- Étudier sa complexité (performance).



## Définition de `liste-images-idem`

```
;;; liste-images-idem : nat * image -> LISTE[image]
;;; (liste-images-idem n image) rend une liste de n images
;;; identiques à l'image donnée
(define (liste-images-idem n image)
  (if (> n 0)
      (cons image (liste-images-idem (- n 1) image))
      (list)))
```





## Spécification de `superposition`

Une illustration sur un autre exemple :

Superposer sur une seule image toutes les images d'une liste

```
;;; superposition : LISTE[image] -> image
;;; (superposition L-image) rend l'image constituee
;;; de la superposition de toutes les images de L-image
```

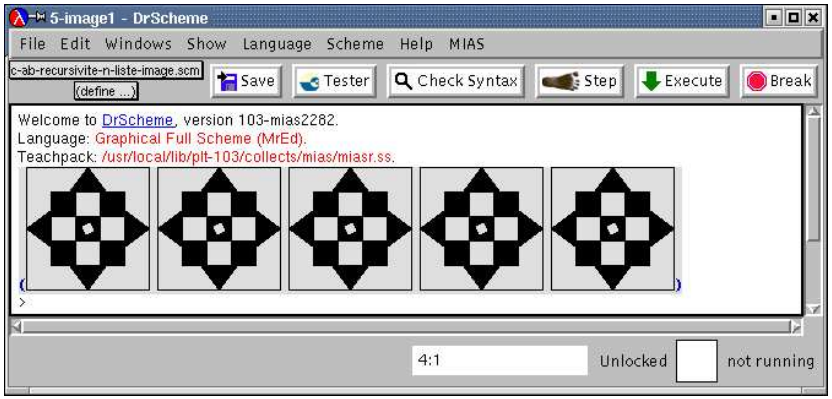
## DÉFINITION DE superposition

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



```
(let ((i1 (filled-triangle 0 1 0 0.6 -0.3 0.6))
      (i2 (filled-triangle 0 -0.2 -0.2 -0.2 -0.2 0.2))
      (i3 (filled-triangle -0.2 0.2 -0.6 0.2 -0.6 0.6))
      (i4 (filled-triangle 0.2 0.2 0.6 0.2 0.6 0.6))
      (i5 (filled-triangle 0 -1 0 -0.6 -0.3 -0.6)))
  (liste-images-idem 5 (4-rotations
                        (superposition
                         (list i1 i2 i3 i4 i5)))))
```










## ANNONCES DIVERSES

- **Le devoir sur table** : le mercredi 19 novembre de 18h30 à 20h
- Les différents soutiens :
  - TME de soutien
  - Permanence à l'Utes
  - Tutorat le mercredi de 9h à 10h30
  - Le forum de WebCT
- Consulter le site

http://www.infop6.jussieu.fr/deug/2003/mias/mias-a/