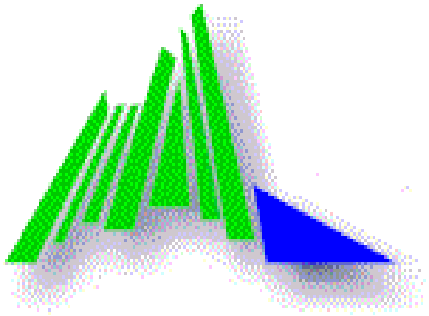
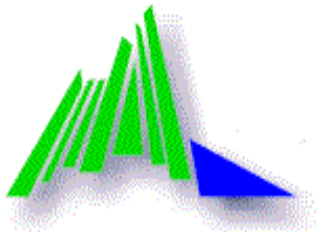


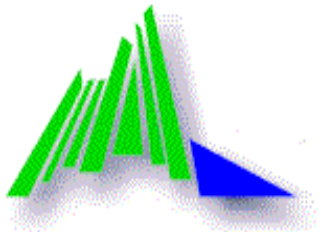
Récursivité



Récursivité et fonction récursive

- Similaire à la récurrence mathématique : si on sait résoudre un problème pour une ou plusieurs tailles assez petites ; si de plus on sait, connaissant la solution pour une taille, résoudre pour une taille supérieure, alors on sait résoudre le problème à toutes les tailles.

- En informatique, une fonction *f* est dite récursive si son corps fait référence à elle-même. C'est-à-dire que l'expression qui constitue son corps contient au moins une application de *f* (appel récursif).

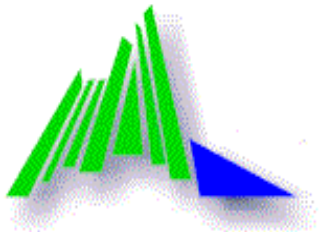


Récursivité et fonction récursive

- **Variable de récursivité (ou d'induction)** : un paramètre de la fonction qui est utilisé pour mettre en œuvre et contrôler le mécanisme récursif (il peut y en avoir plusieurs).

- Le corps de la fonction doit tester la valeur de cette variable. Deux cas possibles :
  - Cas de base : valeur limite connue pour laquelle le résultat peut être calculé directement sans appel récursif.

    ☞ Appel récursif terminal ou branche terminale
  - C'est une valeur quelconque, pour laquelle le résultat n'est pas calculable directement. Il faut alors effectuer un appel récursif en faisant varier la variable de récursivité pour atteindre une des valeurs limites connues et se ramener ainsi à un cas de base.



Exemples de fonctions récursives

```
(define (factorielle n)
    (if (zero? n)
        1
        (* n (factorielle (- n 1)))))
```

```
(define (puissance x n)
    (if (zero? n)
        1
        (* x (puissance x (- n 1)))))
```

Exemples de fonctions récursives

```
(define (factorielle n)
    (if (zero? n)
        1
        (* n (factorielle (- n 1)))))
```

Variable de récursivité

```
(define (puissance x n)
    (if (zero? n)
        1
        (* x (puissance x (- n 1)))))
```

Exemples de fonctions récursives

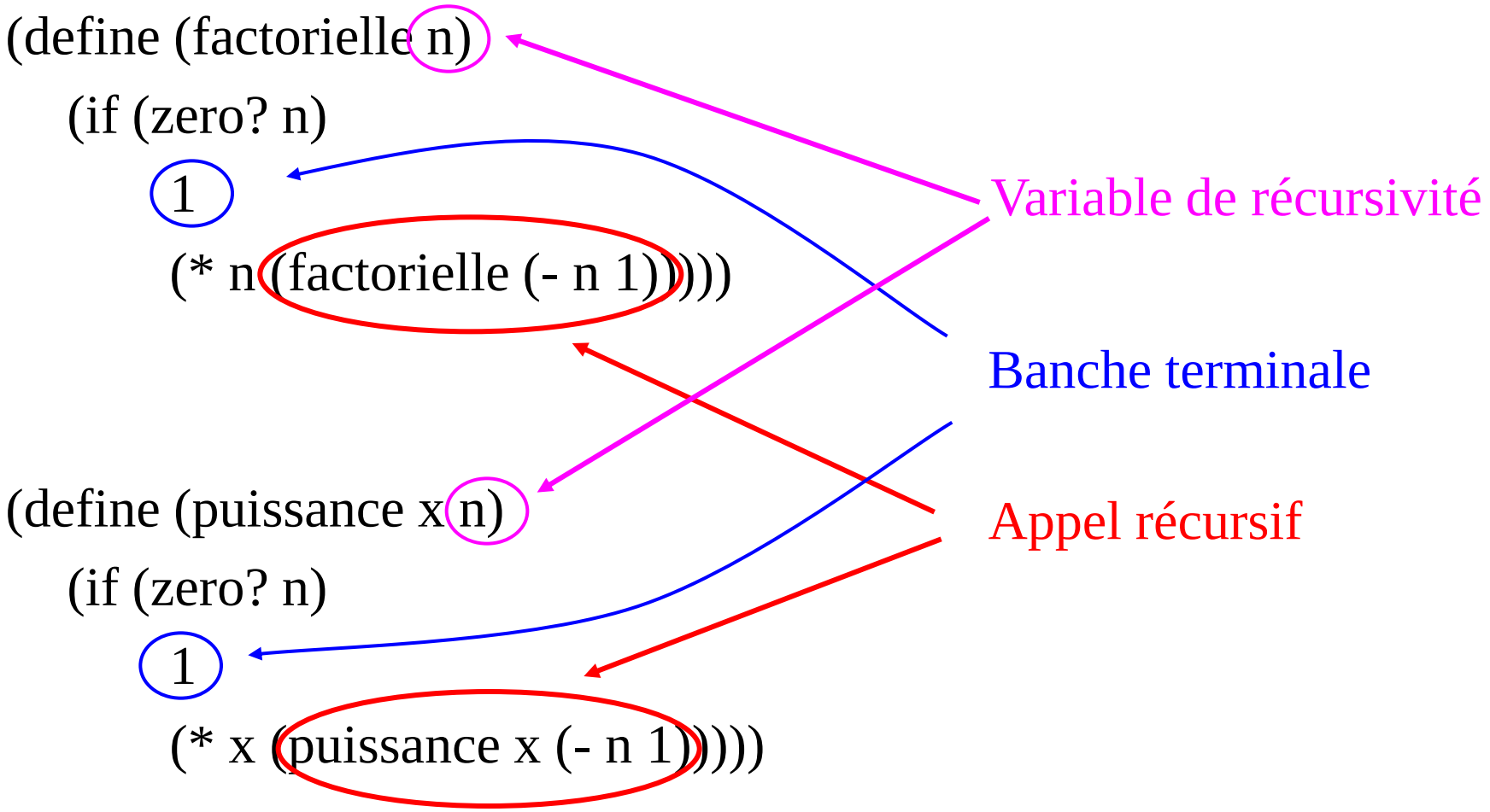
```
(define (factorielle n)
    (if (zero? n)
        1
        (* n (factorielle (- n 1)))))
```

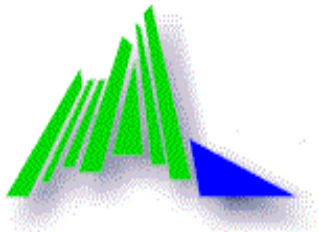
Variable de récursivité

Banche terminale

```
(define (puissance x n)
    (if (zero? n)
        1
        (* x (puissance x (- n 1)))))
```

Exemples de fonctions récursives



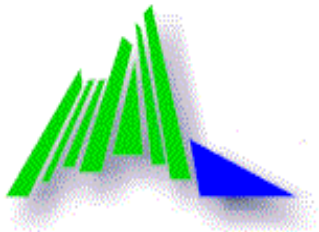


Le mécanisme récursif

```
(define (factorielle n)
    (if (zero? n)
          1
          (* n (factorielle (- n 1)))))
```

- Pour calculer *factorielle (n)* : calculer *factorielle (n-1)*
- Pour calculer *factorielle (n-1)* : calculer *factorielle (n-2)*
- Pour calculer *factorielle (n-2)* : calculer *factorielle (n-3)*
- …
- Pour calculer *factorielle (1)* : calculer *factorielle (0)*

- factorielle (0) est connu, à partir de là tous les autres calculs peuvent se faire. C'est l'appel récursif terminal qui permet de terminer le processus.

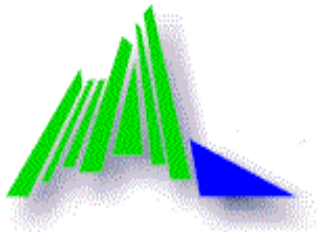


Le mécanisme récursif

```
(define (factorielle n)
    (if (zero? n)
        1
        (* n (factorielle (- n 1)))))
```

- Notion de pile : structure mémoire pour mémoriser les calculs organisée comme une pile de calculs.
- Exemple : calcul de *factorielle (5)*

pile { (* 5 (factorielle 4))



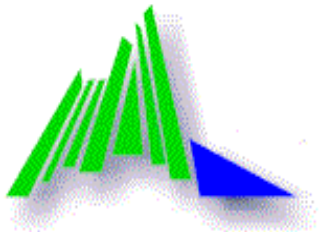
Le mécanisme récursif

```
(define (factorielle n)
    (if (zero? n)
        1
        (* n (factorielle (- n 1)))))
```

- Notion de pile : structure mémoire pour mémoriser les calculs organisée comme une pile de calculs.
- Exemple : calcul de *factorielle (5)*

pile

```
( * 5 ( factorielle 4))
    (* 4 (factorielle 3))
```



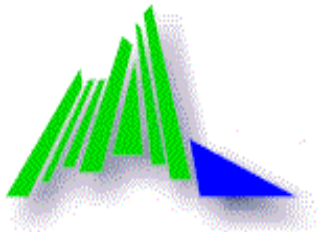
Le mécanisme récursif

```
(define (factorielle n)
    (if (zero? n)
        1
        (* n (factorielle (- n 1)))))
```

- Notion de pile : structure mémoire pour mémoriser les calculs organisée comme une pile de calculs.
- Exemple : calcul de *factorielle (5)*

pile

```
( * 5 ( factorielle 4))
    (* 4 (factorielle 3))
        (* 3 (factorielle 2))
```



Le mécanisme récursif

```
(define (factorielle n)
    (if (zero? n)
        1
        (* n (factorielle (- n 1)))))
```

- Notion de pile : structure mémoire pour mémoriser les calculs organisée comme une pile de calculs.
- Exemple : calcul de *factorielle (5)*

pile

```
( * 5 ( factorielle 4))
     (* 4 (factorielle 3))
          (* 3 (factorielle 2))
               (* 2 (factorielle 1))
```

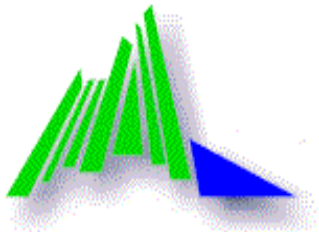
Le mécanisme récursif

```
(define (factorielle n)
    (if (zero? n)
            1
            (* n (factorielle (- n 1)))))
```

- Notion de pile : structure mémoire pour mémoriser les calculs organisée comme une pile de calculs.
- Exemple : calcul de *factorielle (5)*

pile

```
( * 5 ( factorielle 4))
      (* 4 (factorielle 3))
            (* 3 (factorielle 2))
                  (* 2 (factorielle 1))
                        (* 1 (factorielle 0))
```



Le mécanisme récursif

```
(define (factorielle n)
    (if (zero? n)
        1
        (* n (factorielle (- n 1)))))
```

- Notion de pile : structure mémoire pour mémoriser les calculs organisée comme une pile de calculs.
- Exemple : calcul de *factorielle (5)*

pile

```
( * 5 ( factorielle 4))
      (* 4 (factorielle 3))
            (* 3 (factorielle 2))
                  (* 2 (factorielle 1))
                        (* 1 (factorielle 0))    Appel récursif terminal
                              1
```

Le mécanisme récursif

```
(define (factorielle n)
    (if (zero? n)
        1
        (* n (factorielle (- n 1)))))
```

- Notion de pile : structure mémoire pour mémoriser les calculs organisée comme une pile de calculs.
- Exemple : calcul de *factorielle (5)*

pile

```
( * 5 ( factorielle 4))
      (* 4 (factorielle 3))
            (* 3 (factorielle 2))
                  (* 2 (factorielle 1))
                        1
```

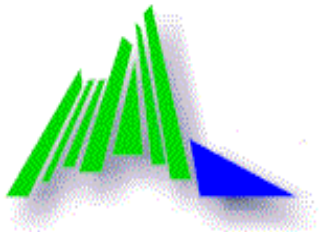
Le mécanisme récursif

```
(define (factorielle n)
    (if (zero? n)
            1
            (* n (factorielle (- n 1)))))
```

- Notion de pile : structure mémoire pour mémoriser les calculs organisée comme une pile de calculs.
- Exemple : calcul de *factorielle (5)*

pile

```
( * 5 ( factorielle 4))
      (* 4 (factorielle 3))
            (* 3 (factorielle 2))
                  2
```



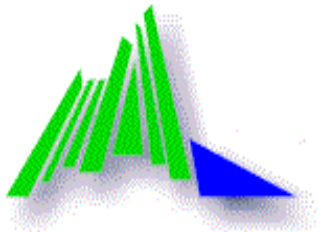
Le mécanisme récursif

```
(define (factorielle n)
    (if (zero? n)
        1
        (* n (factorielle (- n 1)))))
```

- Notion de pile : structure mémoire pour mémoriser les calculs organisée comme une pile de calculs.
- Exemple : calcul de *factorielle (5)*

pile

```
( * 5 ( factorielle 4))
      (* 4 (factorielle 3))
            6
```



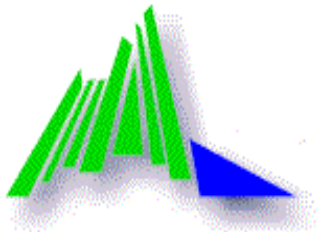
Le mécanisme récursif

```
(define (factorielle n)
    (if (zero? n)
        1
        (* n (factorielle (- n 1)))))
```

- Notion de pile : structure mémoire pour mémoriser les calculs organisée comme une pile de calculs.
- Exemple : calcul de *factorielle (5)*

pile

```
( * 5 ( factorielle 4))
    24
```

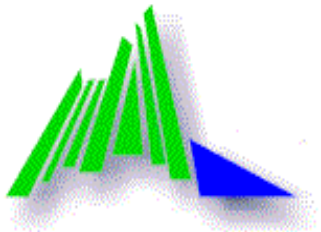


Le mécanisme récursif

```
(define (factorielle n)
    (if (zero? n)
        1
        (* n (factorielle (- n 1)))))
```

- Notion de pile : structure mémoire pour mémoriser les calculs organisée comme une pile de calculs.
- Exemple : calcul de *factorielle (5)*

pile { 120



Le mécanisme récursif

```
(define (factorielle n)
    (if (zero? n)
        1
        (* n (factorielle (- n 1)))))
```

- Notion de pile : structure mémoire pour mémoriser les calculs organisée comme une pile de calculs.
- Exemple : calcul de *factorielle (5)*

  120

Trace de l'exécution d'une fonction récursive

- Fonction trace : permet de visualiser les étapes du mécanisme récursif

- Sous DrRacket
  1. Charger la librairie trace : (#%require racket/trace)

  2. Activer le processus de trace d'une fonction *f* : (trace *f*)
     (trace factorielle)

  3. Appliquer la fonction:
     (factorielle 5)

  4. Désactiver le processus de trace : (untrace *f*)
     (untrace factorielle)

```
|(factorielle 5)
| (factorielle 4)
| |(factorielle 3)
| | (factorielle 2)
| | |(factorielle 1)
| | | (factorielle 0)
| | | 1
| | |1
| | 2
| |6
| 24
|120
```

Ecriture d'une fonction récursive

- Considérons la fonction **somme-carres** qui associe à un entier n la somme des carrés des entiers de 1 à n

  somme-carres (n) = $1 + 2^2 + 3^2 + ... + (n-1)^2 + n^2$

Ecriture d'une fonction récursive

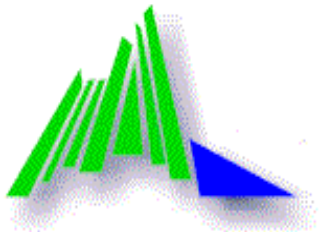
- Considérons la fonction **somme-carres** qui associe à un entier n la somme des carrés des entiers de 1 à n

  somme-carres (n) = $1 + 2^2 + 3^2 + ... + (n-1)^2 + n^2$

- En regroupant les n-1 premiers termes, on en déduit que cette fonction vérifie la relation de récurrence :

  somme-carres(n) = somme-carres(n-1) + n^2

  avec la convention que somme-carres(0) = 0.



Ecriture d'une fonction récursive

- Considérons la fonction **somme-carres** qui associe à un entier n la somme des carrés des entiers de 1 à n

  somme-carres (n) = $1 + 2^2 + 3^2 + ... + (n-1)^2 + n^2$

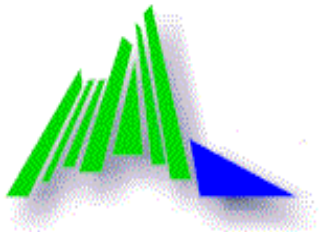
- En regroupant les n-1 premiers termes, on en déduit que cette fonction vérifie la relation de récurrence :

  somme-carres(n) = somme-carres(n-1) + n^2

  avec la convention que somme-carres(0) = 0.

- On peut donc écrire en scheme :

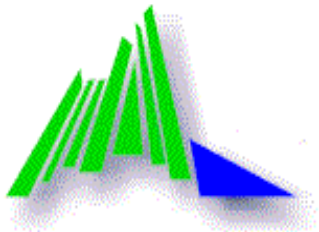
```
(define (somme-carres n)
    (if  (equal? n 0)
         0
         (+ (somme-carres (- n 1)) (expt n 2))))
```



Ecriture d’une fonction récursive

- L’écriture de fonction récursive ne concerne pas seulement les fonctions définies par une relation de récurrence. C'est une méthode générale et très puissante d'expression et de résolution de problèmes.

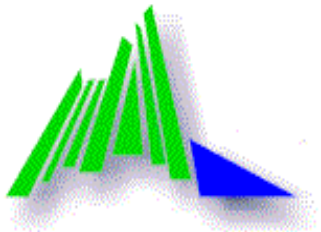
- D'un point de vue plus général, dans une approche purement fonctionnelle on utilise la récursivité pour répéter un traitement (c'est-à-dire l'application d'une même fonction).



Ecriture d'une fonction récursive

- Exemple : écrire une boucle avec un menu

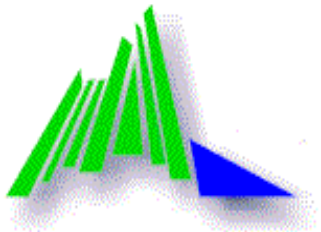
```
( define (menu)
   (begin
    (display "taper 0 pour quitter, 1 pour l'action1, 2 pour l'action2 :")
    (let ((lu (read)))
    (cond
      ((equal? lu 0) (display "au revoir"))
      ((equal? lu 1) action1 (newline) (menu))
      ((equal? lu 2) action2 (newline) (menu))
      (else (display "commande inconnue") (newline) (menu))))))
```



Ecriture d'une fonction récursive : quelques règles

1. Parmi tous les paramètres, quel est celui qui caractérise la taille du problème ? Ce sera **la variable de récursivité** *vr*
2. Pour quelle valeur de *vr* est-il possible de résoudre le problème de façon triviale ? Appelons cette valeur *val_init*.
3. Quelle est la solution au problème si *vr* vaut *val_init*? Ce sera le résultat de **la branche terminale** de la récursivité.
4. Comment faire varier *vr* pour que ses valeurs successives se rapprochent de plus en plus de *val_init* ? Ce sera **la fonction de variation**.
5. Si l'on connaît la solution du problème pour une taille quelconque, quel calcul simple faut-il réaliser avec cette solution pour résoudre la problème pour la taille supérieure ou inférieure selon la fonction de variation de *vr*.

☞ **Une fonction récursive doit toujours posséder au moins une branche terminale pour garantir la terminaison des appels récursif. Sinon risque de boucle infinie.**



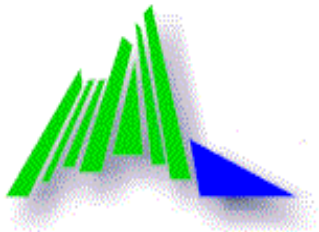
Processus itératif vs processus récursif

- Il faut distinguer :
  - écriture (définition) récursive d'une fonction
  - processus (exécution) récursif

- Processus récursif : Mémorisation des calculs en attente dans une pile. Les calculs sont effectués après l'évaluation d'une branche terminale de la récursivité. Le résultat est obtenu à la fin.

- Processus itératif : Les calculs sont effectués au fur et à mesure. Le résultat est obtenu lors de l'évaluation d'une branche terminale.

- Une fonction récursive peut s'exécuter selon un processus récursif ou selon un processus itératif (on parle alors de récursivité terminale).



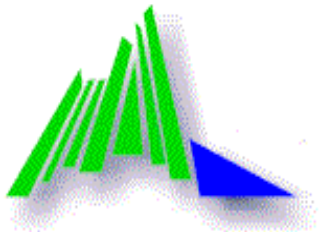
Processus itératif vs processus récursif

- Ecrire la fonction *rebours* qui prend en paramètre un nombre *n*, et qui affiche à l'écran :

```
n
n-1
…
2
1
0
```

- Ecrire la fonction *compte* qui prend en paramètre un nombre *n*, et qui affiche à l'écran :

```
0
1
2
…
n-1
n
```



Processus itératif vs processus récursif

- Ecrire la fonction *rebours* qui prend en paramètre un nombre *n*, et qui affiche à l'écran :

```
(define (rebours n)
  (if (zero? n)
       (begin
           (display 0)
           (newline))
       (begin
           (display n)
           (newline)
           (rebours (- n 1)))))
```

- Ecrire la fonction *compte* qui prend en paramètre un nombre *n*, et qui affiche à l'écran :

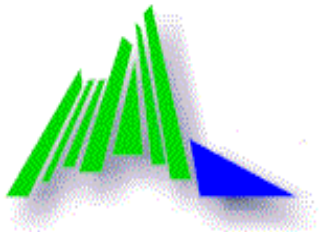
```
(define (compte n)
  (if (zero? n)
      (begin
         (display 0)
         (newline))
      (begin
        (compte (- n 1))
        (display n)
        (newline))))
```

Exemple fonction récursive/processus récursif

- Les tours de hanoï

  - La légende dit que dans un temple d'Hanoï il y a trois tours.
  - A l'origine, il y avait 64 disques, empilés du plus grand au plus petit, sur une des tours.
  - A chaque fois qu'ils viennent prier, les moines déplacent un disque du sommet d'une tour à une autre en respectant la règle suivante : on ne doit pas poser un disque sur un autre plus petit.
  - Quand tous les disques auront été empilés sur une autre des tours, ce sera la fin du monde.
  - Bien que personne ne souhaite la fin du monde, le problème est de transférer la pile de disque d'une tour sur une autre en un minimum de mouvements.

Voir http://fr.wikipedia.org/wiki/Tours_de_Hanoï

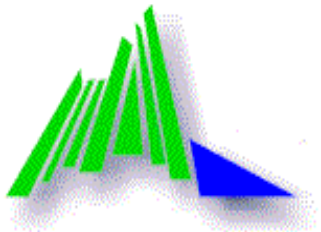


Exemple fonction récursive/processus récursif

- Considérons que les trois tours sont numérotées t1, t2 et t3 de gauche à droite. La méthode pour déplacer n disques de la tour t1 à la tour t3 est la suivante :
  - déplacer n-1 disques de la tour t1 à la tour t2,
  - déplacer le dernier disque de la tour t1 à la tour t3 (mouvement élémentaire),
  - déplacer n-1 disques de la tour t2 à la tour t3.
- Ecrire une fonction récursive *hanoi (n, t1, t2, t3)* qui affiche tous les déplacements sous la forme : *déplacement de* t_i *vers* t_j
- Exemple pour n = 3

(hanoi 3 't1 't2 't3)

déplacement de t1 vers t3
déplacement de t1 vers t2
déplacement de t3 vers t2
déplacement de t1 vers t3
déplacement de t2 vers t1
déplacement de t2 vers t3
déplacement de t1 vers t3



Exemple fonction récursive/processus récursif

```
(define (hanoi1 n t1 t2 t3)
  (if (= n 1)
      (begin
        (display "déplacement de ")
        (display t1)
        (display " vers ")
        (display t3)
        (newline)
        )
      (begin
        (hanoi1 (- n 1) t1 t3 t2)
        (hanoi1 1 t1 t2 t3)
        (hanoi1 (- n 1) t2 t1 t3)
        )))
```

```
(hanoi1 3 't1 't2 't3)
|(hanoi1 3 t1 t2 t3)
| (hanoi1 2 t1 t3 t2)
| |(hanoi1 1 t1 t2 t3)
déplacement de t1 vers t3
| |#<void>
| |(hanoi1 1 t1 t3 t2)
déplacement de t1 vers t2
| |#<void>
| (hanoi1 1 t3 t1 t2)
déplacement de t3 vers t2
| #<void>
| (hanoi1 1 t1 t2 t3)
déplacement de t1 vers t3
| #<void>
|(hanoi1 2 t2 t1 t3)
| (hanoi1 1 t2 t3 t1)
déplacement de t2 vers t1
| #<void>
| (hanoi1 1 t2 t1 t3)
déplacement de t2 vers t3
| #<void>
|(hanoi1 1 t1 t2 t3)
déplacement de t1 vers t3
|#<void>
```