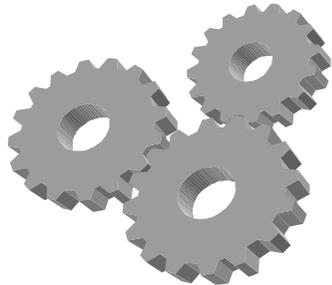




# Compléments sur la Récurrence

## - Les Calculs Itératifs -



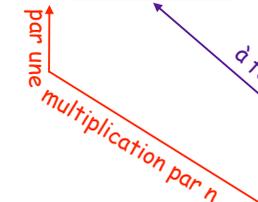
p. 101-112

## La récurrence enveloppée (la "vraie")

```
(define (fac n) ; n entier ≥ 0
  (if (= 0 n)
      1
      (* (fac (- n 1)) n)))
```

$n! = (n - 1)! \times n$

```
(fac 4)
= (* (fac 3) 4)
= (* (* (fac 2) 3) 4)
= (* (* (* (fac 1) 2) 3) 4)
= (* (* (* (* (fac 0) 1) 2) 3) 4)
= (* (* (* (* 1 1) 2) 3) 4)
= (* (* (* 1 2) 3) 4)
= (* (* 2 3) 4)
= (* 6 4)
= 24
```



L'appel récursif étant **enveloppé**, il faut mémoriser des calculs en attente !

L'**enveloppe** est la fonction :  
 $(\lambda (r) (* r n))$

Je mets la multiplication par n en attente et je calcule (n-1)!

expansion-contraction...

## La récurrence terminale ou ITERATION

- Le schéma de calcul récursif précédent n'est pas le seul ! Faisons un peu d'*algèbre* sur ce programme. Regardons de près l'appel récursif :

```
(* (fac (- n 1)) n)
```

- Généralisons-le** en perdant un peu d'information :

```
(* (fac p) q)
```

- Introduisons une fonction auxiliaire à deux paramètres :

```
(aux p q) = (* (fac p) q)
```

- Cherchons une définition intrinsèque de la fonction aux :

```
p = 0 → (aux 0 q) = (* (fac 0) q) = (* 1 q) = q
p ≠ 0 → (aux p q) = (* (fac p) q) = (* (* p (fac (- p 1)) q)
= (* (fac (- p 1)) (* p q))
= (aux (- p 1) (* p q))
```

in English : "Tail Recursion"...

Bingo !

- Nous disposons donc d'une définition récursive de aux :

```
(define (aux p q)
  (if (zero? p)
      q
      (aux (- p 1) (* p q))))
```

- Comparons le schéma de calcul obtenu avec le précédent :

```
(fac 4)
= (aux 4 1)
= (aux 3 4)
= (aux 2 12)
= (aux 1 24)
= (aux 0 24)
= 24
```

Plus de phénomène d'expansion-contraction !

- La fonction aux appelle directement la fonction aux. On dit que l'appel récursif est en **position terminale**, ou **non enveloppé**.

**DEFINITION : Une fonction récursive est dite itérative si son appel récursif est en position terminale.**

## Qu'est-ce qu'une boucle ?

• Dans le schéma de calcul précédent (itératif), on voit que les **variables de boucle** p et q sont mises à jour à chaque étape. Une condition de sortie précise la fin de ce processus.

① Si une certaine condition sur p et q est remplie, le résultat est ...

```
(define (aux p q)
  (if (zero? p)
      q
      .....))
```

② Sinon, on itère le calcul en mettant à jour les variables de boucles.

```
(define (aux p q)
  (if (zero? p)
      .....
      (aux (- p 1) (* p q))))
```

• Le schéma le plus simple d'une **boucle** f est donc en théorie :

$$f(x_1, \dots, x_n) = \text{SI } g(x_1, \dots, x_n) \text{ ALORS res SINON } f(x'_1, \dots, x'_n) \quad 5$$

• En conséquence de quoi, il est sain de **localiser la fonction auxiliaire au sein de la fonction principale**, pour la cacher à l'utilisateur :

```
(define (fac n) ; la fonction principale
  (local [(define (aux p q)
            (if (zero? p)
                q
                (aux (- p 1) (* p q))))])
  (aux n 1)))
```

```
> (fac 5)
120
> (aux 120 1)
```

*reference to an identifier before its definition: aux*

• Par extension on dit que la fonction fac est itérative, puisque son processus de calcul est pris en charge par la fonction aux, qui l'est.

• Pouvoir coder des **sous-fonctions** privées à l'intérieur d'une fonction est un mécanisme logiciel sain, disponible en Python, Javascript, Ruby, etc. mais pas de manière simple en C ou en Java hélas.

## Localiser les fonctions intermédiaires

• Une fonction itérative est la plupart du temps rédigée sous la forme de deux fonctions :

- la fonction elle-même, qui fait immédiatement appel à une autre fonction auxiliaire :

```
(define (fac n) ; la fonction principale
  (aux n 1))
```

- la fonction auxiliaire qui implémente la boucle :

```
(define (aux p q) ; la cheville ouvrière
  (if (zero? p)
      q
      (aux (- p 1) (* p q))))
```

• L'utilisateur n'utilisera que la fonction fac. Il n'a aucune raison de connaître l'existence de la fonction auxiliaire !

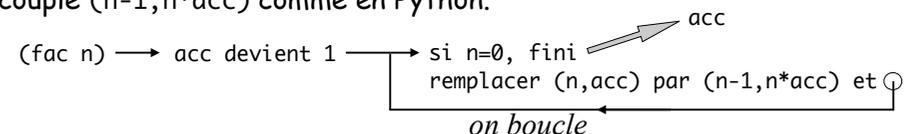
• Enfin, il est usuel de nommer **iter** une fonction locale qui implémente une boucle. Ici, p joue le rôle de n et q celui d'un accumulateur, qu'il est usuel de nommer acc :

```
(define (fac n) ; la fonction principale
  (local [(define (iter n acc)
            (if (zero? n)
                acc
                (iter (- n 1) (* n acc))))])
  (iter n 1)))
```

*Version finale*

*j'itère en remplaçant en même temps n par n-1 et acc par n\*acc*

• Insistons sur le fait que les deux remplacements sont effectués *en même temps*. On devrait dire : je remplace le couple (n, acc) par le couple (n-1, n\*acc) comme en Python.



- Si l'on veut **prouver** que ce programme calcule bien une factorielle, il est important de pouvoir dire ce que calcule la fonction (iter n acc).

```
(define (iter n acc) ; calcule n! * acc
  (if (zero? n)
      acc
      (iter (- n 1) (* n acc))))
```

**THEOREME** : j'affirme que (iter n acc) calcule  $n! \times acc$ .

**PREUVE** : Récurrence sur  $n \geq 0$ . Si  $n=0$ , (iter 0 acc) = acc =  $0! \times acc$  donc c'est vrai pour  $n=0$ . Supposons  $n > 0$  et l'assertion vérifiée pour  $n-1$ . Alors :

```
(iter n acc) = (iter (- n 1) (* n acc))
              = (* (fac (- n 1)) (* n acc)) par HR
              = (* (* (fac (- n 1)) n) acc) par associativité
              = (* (fac n) acc) d'où la récurrence
```

**COROLLAIRE** : j'affirme que (fac n) calcule  $n!$

**PREUVE** : (fac n) = (iter n 1) =  $n! \times 1 = n!$

9

## Il y aurait donc deux modèles de calcul ?

- Face à la tâche de programmer une fonction :
  - Essayez d'abord de **raisonner par récurrence**, en envisageant tous les cas possibles, en partant du cas le plus simple sur lequel tous les autres cas vont converger. C'est souvent le plus facile :

```
(define ($xpt x n) ; n ≥ 0, calcule x^n
  (cond ((= n 0) ...)
        ((even? n) ...) ; dichotomie !
        (else ...)))
```

- En cas d'échec, ou si la solution n'est pas assez rapide, ou si vous pensez tenir un **schéma itératif avec vases communicants**, essayez de programmer une itération en introduisant une fonction auxiliaire :

```
(define ($xpt x n) ; n ≥ 0, calcule x^n itérativement
  (local [(define (iter ...) ; boucle + dichotomie
            ...)] ; un peu plus difficile...
    (iter ...)))
```

11

## Et l'interprétation de tout ça ?

- Il faut bien se pénétrer du schéma itératif, qui modélise la plupart du temps un phénomène de **vases communicants**.

- Une variable A se vide, une autre variable B se remplit.

- Lorsque A est complètement vide, B est le résultat ou presque.

- On dit que B est un **accumulateur**.

## Et l'intérêt de tout ça ?

- Programmer un algorithme **itératif**

[ $\Leftrightarrow$  sous forme de boucle] permet :

- de pouvoir à chaque étape tenir en main le résultat en cours de construction, ici acc. Cela peut parfois servir...

- d'accélérer *un peu* l'exécution, en évitant les calculs en attente.

A	B
n	acc
5	1
4	5
3	20
2	60
1	120
0	120

(fac 5)

10

## L'un des modèles est-il meilleur que l'autre ?

- Aucun des deux n'est plus puissant au sens où lui seul pourrait effectuer tel ou tel calcul. Si une fonction est calculable, on a le choix de procéder ou non par itération.

- Mais ce choix est purement théorique. Seule l'expérience des problèmes montre comme vite trouver l'angle d'attaque. C'est pareil pour toutes les sciences, les maths notamment !

- En règle générale, **la récurrence est d'une puissance étonnante**. Ne vous forcez pas à produire une itération à tout prix.

- Ne croyez pas les bonnes âmes qui expliquent que l'itération est plus rapide. Ce qui compte c'est la **COMPLEXITE** de votre fonction !

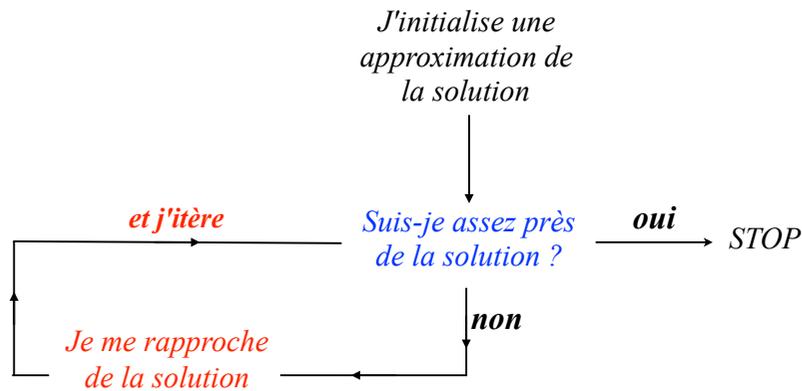
- Si vous avez déjà programmé, n'opposez pas la récurrence à l'itération ! Celle-ci est un cas particulier de récurrence [terminale]. Au moins dans les langages [comme Scheme] qui **optimisent** la récurrence...

12

## Quelques résolutions itératives de problèmes

- Cas typique d'un **processus itératif**. Mon but est de trouver une **solution approchée à un problème** :

- Si je suis assez proche de la solution, je m'en contente !
- Sinon, j'essaye de me rapprocher de la solution, et j'itère.



13

- Mon approximation courante  $a$  est-elle **assez bonne** ? Elle est assez bonne lorsque  $a^2$  est proche de  $r$ , donc lorsque  $|a^2 - r|$  est proche de 0. Notons  $h$  le paramètre de précision, par exemple  $h = 0.001$ .

$a$  est assez bonne dès que  $|a^2 - r| < h$

```
(define (assez-bonne? a r h)
  (< (abs (- (sqr a) r)) h))
```

- Pour **améliorer** l'approximation, il suffit d'appliquer la formule de Newton, qui fait converger vers  $\sqrt{r}$  :

```
(define (ameliore a r)
  (* 1/2 (+ a (/ r a))))
```

- Une **boucle de calcul** permet enfin d'**itérer** l'amélioration jusqu'à ce qu'elle soit assez bonne :

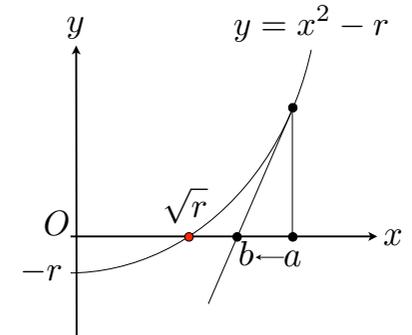
```
(define (iter a r h)
  (if (assez-bonne? a r h)
      a
      (iter (ameliore a r) r h)))
```

15

## Calcul d'une racine carrée par la méthode de Newton

- Cf. livre d'Abelson & Sussman [à la B.U.], chapitre 1. Soit à calculer la **racine carrée approchée d'un nombre réel  $r > 0$** .

- **NEWTON** : si  $a$  est une approximation de  $\sqrt{r}$  alors  $b = \frac{1}{2}(a + \frac{r}{a})$  est une approximation encore meilleure !



- Justification : passer par les tangentes, la méthode est assez générale...
- Nous allons développer cet algorithme à-travers plusieurs fonctions.

- mon approximation courante  $a$  est-elle *assez bonne* ?
- comment *améliorer* l'approximation ?
- comment faire converger itérativement le processus ?

14

- Il reste à **soigner la présentation** et à **localiser** les fonctions auxiliaires, ce qui au passage élimine les paramètres inutiles :

```
; Calcul de la racine carrée approchée de  $r > 0$ ,
;  $h$  gouverne la précision, et  $a$  est l'approximation initiale.
```

```
(define (rac2 r a h)
  (local [(define (iter a)
            (if (assez-bonne? a)
                a
                (iter (ameliore a))))
          (define (assez-bonne? a)
            (< (abs (- (sqr a) r)) h))
          (define (ameliore a)
            (* #i0.5 (+ a (/ r a))))])
    (iter a)))
```

```
> (rac2 2 10 0.1)
#i1.4442380948662321
> (rac2 2 10 0.01)
#i1.4145256551487377
> (rac2 2 10 1e-3)
#i1.4145256551487377
```

```
> (rac2 2 10 1e-4)
#i1.4142135968022693
> (rac2 2 10 1e-8)
#i1.4142135623730954
> (sqrt 2)
#i1.4142135623730951
```

← En 7 tours de boucle !!

16

## Le PGCD à la Euclide

- Soient  $a$  et  $b$  deux entiers naturels,  $b > 0$ . Alors :

*Le PGCD de  $a$  et  $b$  est le même que celui de  $b$  et du reste de la division de  $a$  par  $b$  [Euclide, Eléments, Livre VII]*

- Exemple :  $\text{pgcd}(8, 12) = \text{pgcd}(12, 8) = \text{pgcd}(8, 4) = \text{pgcd}(4, 0)$  STOP !
- STOP car le Plus Grand Commun Diviseur de 4 et de 0, c'est 4.
- Et le second argument finira toujours par devenir égal à 0 car le passage de  $b$  au reste  $r$  de la division de  $a$  par  $b$  est strictement décroissant. En effet :  $a = bq + r$ , avec  $0 \leq r < b$ .

```
(define (pgcd a b) ; a et b ∈ N
  (if (zero? b)
      a
      (pgcd b (modulo a b))))
```

- L'algorithme est **spontanément itératif** : la récurrence est terminale ↓

• ATTENTION : Ne croyez surtout pas que le **passage d'une récurrence enveloppée à une récurrence terminale** [itération] suffit à faire baisser la complexité ! **En général la complexité reste la même...**

• Forcer une fonction à être itérative revient à éliminer la mise en attente de calculs intermédiaires. Ceci économise un peu d'espace [de pile] mais n'est pas un gage d'optimisation drastique ! Concentrez-vous plutôt sur la **complexité** de vos algorithmes...

• Les langages traditionnels [Python, Java par exemple] sont obligés d'avoir des mots spéciaux pour exprimer les itérations : while, for, etc. Et cela pour pallier à une insuffisance : **dans ces langages la récurrence n'est pas optimisée, on doit l'éviter si possible !**

```
(define (pgcd a b) ; a et b ∈ N
  (if (zero? b)
      a
      (pgcd b (modulo a b))))
```

**Scheme**  
(itératif, fonctionnel)

```
def pgcd (a, b) : # a et b ∈ N
  while b != 0 :
    (a, b) = (b, a % b)
  return a
```

**Python**  
(itératif, impératif)



19

## L'inversion d'une liste

- Nous avons vu au cours 6 page 14 un algorithme pour ( $\text{reverse } L$ ) dont la **complexité** était en  $O(n^2)$  à cause de l'utilisation de append.

- Or à la main, nous procéderions par **transfert** dans un accumulateur  $\text{acc}$ , vide au début :

L	acc
(a b c d)	()
(b c d)	(a)
(c d)	(b a)
(d)	(c b a)
()	(d c b a)

```
(define ($reverse L) ; algorithme en O(n)
  (local [(define (iter L acc)
            (if (empty? L)
                acc
                (iter (rest L) (cons (first L) acc))))])
  (iter L empty)))
```

Vases communicants !...

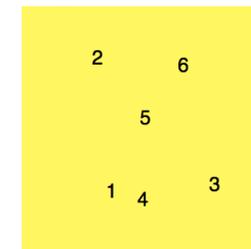
18

## Construction itérative d'une image

- Je me propose de jeter dans un canvas 300 x 300 les chiffres d'un entier, à des positions aléatoires.

- Et de manière itérative : l'accumulateur sera une image !

> (chiffres->image 431526)



```
(define (chiffres->image n) ; n entier > 0
  (local [(define FOND (rectangle 300 300 'solid "yellow"))
          (define (rand) (+ 50 (random 201))) ; dans [50,250]
          (define (iter n acc) ; l'accumulateur est une image
            (if (= n 0)
                acc
                (local [(define u (modulo n 10)) ; le chiffre des unités
                        (define img (text (number->string u) 24 "black"))]
                  (iter (quotient n 10) (place-image img (rand) (rand) acc))))])
  (iter n FOND)))
```

20

## OPTIONNEL : La boucle do fonctionnelle de Scheme

- Il s'agit d'une **boucle purement fonctionnelle, sans mutation** : juste un **sucré syntaxique** masquant une construction classique plus lourde !

```
(do ((var1 init1 step1) (var2 init2 step2) ...)
    (test-fin resultat))
```

```
(define (fac n) ; n entier ≥ 0
  (do ((i n (- i 1)) (acc 1 (* acc i)))
      ((= i 0) acc)))
```

```
(define (fac n) ; itérative classique
  (local [(define (iter i acc)
            (if (= i 0)
                acc
                (iter (- i 1) (* acc i))))])
    (iter i 1)))
```

*LOURD*

```
def fac(n) : # n entier ≥ 0
  (i,acc) = (n,1)
  while i != 0 :
    (i,acc) = (i-1,acc*i)
  return acc
```

*Python*

*On se contente de décrire les variables de boucle, leur initialisation, leur évolution, le cas d'arrêt, et le résultat.*