OPTION PF2 - Automne 2016
Jean-Paul Roy
L2-Info&Maths, Faculté des Sciences

http://deptinfo.unice.fr/~roy

Cours n°2

La mutation en Programmation Impérative

Chapitre 12

Fonctionnel vs Impératif

• La **programmation fonctionnelle** procède par composition de fonctions récursives. La modification de l'état de la machine est implicite, gérée par le compilateur. La récursivité est en général optimisée, et fournit l'itération en cas particulier.

Conception et preuves facilitées, exécution plus lente (?). Lisp, Scheme, Caml, Haskell, ... — *EXPRESSIONS*

• La **programmation impérative** procède par modifications explicites de la mémoire de la machine, à l'aide d'instructions exécutées en séquence, et destinées à modifier [*faire muter*] des variables. L'itération est privilégiée, la récursivité rarement optimisée.

Conception et preuves difficiles, exécution efficace. Pascal, C, Java, ... — *INSTRUCTIONS*

Le séquencement d'instructions avec begin

• Les expressions Scheme ayant un effet de bord [*side effect*] jouent le rôle d'instructions, par exemple :

```
(printf "x=~a\n" x)              ; aucun résultat
(begin (printf "x=~a\n" x) x)    ; résultat : x
```

• La forme spéciale `(begin` e_1 `...` e_n`)` permet de **séquencer les évaluations** des expressions e_1, ..., e_n. Elle jette les résultats des expressions e_1, ..., e_{n-1} et retourne la valeur de e_n.

• En réalité, **begin est implicite partout** sauf dans un IF !

```
(if (> x 0)
    (begin ...)
    (begin ...))
```

```
(define (foo x)
  (...)
  (...)
  (...))
```

begin implicite

• `(when test e1 e2 …) = (if test (begin e1 e2 …) (void))`

L'instruction d'affectation set!

• C'est le premier opérateur de MUTATION : on fait ***muter*** la valeur d'une variable, sans espoir de retour !

`(set!` *symbole expression*`)`

• Il s'agit d'une forme spéciale : `<symbole>` n'est pas évalué !

• **Sémantique** de l'évaluation de `(set! s e)` dans un environnement E :

① on commence par rechercher la première liaison <s,v> du symbole s dans l'environnement E. S'il n'y en a pas, erreur.
② on évalue e dans E, pour obtenir une valeur w.
③ on remplace v par w. Aucun résultat !

```
> (define x 1)
> (define y 2)
```

```
> (let ((z x) (x 3))
     (set! x (+ x 10))
     (set! y (+ y 20))
     (set! z (+ z 1))
     (list x y z))
(13 22 2)
```

```
> (list x y)
(1 22)
```

• La **factorielle impérative**, avec une boucle pure (lourd) :

lourd !

```
(define (fac-imp n)
  (define res 1)
  (define (iter)
    (printf "n=~a f=~a\n" n res)
    (if (zero? n)
        res
        (begin (set! res (* res n))
               (set! n (- n 1))
               (iter))))
  (iter))
```

Python

```
def fac_imp(n) :
  res = 1
  while True :
    print('n=',n,'res=',res)
    if n == 0 :
      return res
    res = res * n
    n = n - 1
```

• Pour ***tracer*** l'exécution, on peut intercaler un `printf` en début de boucle pour suivre l'évolution des variables de boucle :

```
> (fac-imp 4)
n=4 res=1
n=3 res=4
n=2 res=12
n=1 res=24
n=0 res=24
24
```


• Si l'on dispose des boucles ***while*** et ***for*** comme en Python :

```
(define (fac-while n)
  (let ((res 1))
    (while (> n 0)
      (set! res (* res n))
      (set! n (- n 1)))
    res))
```

```
def fac_while(n) :
  res = 1
  while n > 0 :
      res = res * n
      n = n - 1
  return res
```

```
(define (fac-for n)
  (define res 1)
  (for [(k (in-range 1 (+ n 1)))]
    (set! res (* res k)))
  res)
```

```
def fac_for(n) :
 res = 1
 for k in range(1,n+1) :
    res = res * k
 return res
```

N.B. i) Regardez `let` remplacé par un `define` interne plus moderne…
ii) Voir le fichier `for-examples.rkt` pour les boucles `for` de Racket.

Une boucle WHILE en Scheme ???

• La norme Scheme ne prévoit pas de boucle WHILE. Elle y est superflue puisque ***la récursivité terminale est optimisée*** !

• Néammoins, pour la nostalgie ou les traductions rapides de code Python ou Java par exemple, elle a son intérêt. Son format est :

```
(while test expr1 expr2 ...)
```

• Exemple :

```
> (define x 1)
> (while (< x 10)
    (printf "~a" x)
    (set! x (+ x 1)))
123456789
>
```

```
;  >>> x = 1
;  >>> while x < 10 :
;          print(x,end='')
;          x = x + 1
;
; 123456789
```


• Il est clair que la forme `(while t e1 e2 ...)` ne va pas évaluer ses arguments ! Ce doit donc être une MACRO !

• Quelle sera l'expansion ? Une boucle Scheme s'exprime par un *define interne*… mais en faisant abstraction du problème en cours [dans lequel x serait une variable de boucle] :

```
(while (< x 10)
  (display x)
  (set! x (+ x 1)))
```

⟺

```
(let ()
  (define (iter)
    (if (not (< x 10))
        (void)
        (begin (display x)
               (set! x (+ x 1))
               (iter))))
  (iter))
```

Ce n'est que de la restructuration de code !

• L'équivalence s'exprime par **define-syntax** avec une seule règle syntaxique (on pourrait dans ce cas utiliser `define-syntax-rule`) :

```
(define-syntax while
  (syntax-rules ()
    ((while test e1 e2 ...) (let ()
                              (define (iter)
                                (if (not test)
                                    (void)
                                    (begin e1 e2 ... (iter))))
                              (iter)))))
```

expansion

motif

• Le corps de `iter` s'écrit plus simplement :

```
(when test e1 e2 ... (iter))
```

• Encore une fois : bien voir qu'une macro est destinée à ***bluffer le compilateur***. Vous écrivez quelque chose de simple, mais lui voit autre chose de bien plus compliqué [l'expansion] !...

ATTENTION à l'appel par valeur ! *Call by value*

• L'ordre applicatif d'évaluation fonctionne en **appel par valeur** : les valeurs des paramètres sont restaurées en sortie de la fonction !

• Exemple : j'essaie de simuler le mécanisme ++x de C/Java :

```
(define (++ x)
  (set! x (+ x 1))
  x))
```

```
> int x = 2;
> ++x
3
> x
3
```

JdrJava

```
> (define x 1)
> (++ x)
2
> x
1
```

```
> (define y 1)
> (++ y)
2
> y
1
```

Caramba !
Encore râté ?

• Le paramètre x est ***restauré à sa valeur initiale***. Je suis donc <u>obligé</u> de passer par une MACRO :

```
(define-syntax ++
  (syntax-rules ()
    ((++ x) (begin (set! x (+ x 1)) x))))
```

```
> (define x 1)
> (++ x)
2
> x
2
```

```
> (define y 1)
> (++ y)
2
> y
2
```

Effet de bord !

• La plupart des langages actuels (C, Java, Scheme...) fonctionnent uniquement en **appel par valeur**. Ceci est donc <u>très important</u> !

Il y a peut-être une autre solution...

– Une autre solution à quoi ?

– Au fait que l'on ne peut pas modifier un argument numérique passé par valeur. Par exemple le x de la fonction ++ page 10. Peu importe d'ailleurs qu'il soit numérique ou autre...

– Mais comment faire sans passer par une macro ?

– En le mettant dans une boîte (box) qui est **l'unité élémentaire de mémoire mutable** (en réalité un *vecteur mutable* à 1 élément).

– Ok. Un exemple ?

```
(define (push-the-volume! v)
  (define old-v (unbox v))
  (set-box! v (+ old-v 1)))
```

```
> (define vol (box 5))
> (push-the-volume! vol)
> (unbox vol)
6
> vol          ; I am a box !
#&6
> (box? vol)
#t
```

La mutation des structures

• Parmi les fonctions automatiquement créées par `define-struct` se trouvent des fonctions de **mutation des champs** d'une structure, si l'on spécifie la mutabilité de la structure par `#:mutable`.

```
> (define-struct balle (x y couleur) #:mutable #:transparent)
> (define B (make-balle 10 20 "red"))
> B
#(struct:balle 10 20 "red")
> (balle-x B)
10
> (set-balle-x! B 4)
> B
#(struct:balle 4 20 "red")
```

Affichage des champs

Génération des modificateurs

```
> (when (< (balle-x B) 0)
    (set-balle-x! B 0))
```

(if test p q)
(when test e1 e2 ...)

IMPORTANT et PROFOND : la valeur d'une λ-expression

• Procédons à une expérience fine :

```
> (define a 10)
> (define foo
    (let ((a 1000))
      (lambda (x) (+ x a))))
> foo
#<procedure:foo>
> (set! a (+ a 2))
> a
12
> (foo a)   ; repasse-t-on dans le let ?
1012
```

???

FERMETURE

Bien faire la différence avec :

```
(define foo
  (lambda (x)
    (let ((a 1000))
      (+ x a))))
```

La lambda se souvient donc de l'environnement dans lequel elle a été créée !


```
> (lambda (x) (+ x a))
#<procedure>
```

? Variables libres...

La valeur d'une λ-expression est une FERMETURE.

closure

• **La valeur d'une lambda-expression dans un environnement E est une FERMETURE** : un couple <L,E> formé du texte L de la lambda, et d'un pointeur sur E. C'est dans cet **environnement de création** E et non dans l'**environnement d'exécution** que seront cherchées les **valeurs des variables libres** + et a de la lambda !

• Une fonction définie en Scheme [dont la valeur est une **fermeture**] dispose donc d'une **mémoire privée** : son environnement de création ! Hum, on va pouvoir s'en servir...

Variables privées et générateurs

• On veut programmer un **générateur de nombres pairs**.

• Il faut un compteur n. Mais PAS EN VARIABLE GLOBALE : restons propres !

```
> (define g (make-gpair))
> (g)
0
> (g)
2
> (g)
4
```

etc.

• Euréka : utiliser **l'environnement de création d'une fermeture** pour maintenir des **variables privées** !

• La fonction `(make-gpair)` ci-dessous retourne un *générateur*, qui est une fonction sans paramètre :

```
(define (make-gpair)
  (define n -2)        ; variable privée !
  (lambda ()
    (set! n (+ n 2))
    n))
```

• On peut définir plusieurs générateurs indépendants :

```
> (define fermat (make-gpair))
> (define gauss (make-gpair))
> (fermat)
0
> (fermat)
2
> (fermat)
4
> (gauss)
0
> (gauss)
2
> (fermat)
6
> (gauss)
4
> n
ERROR : undefined identifier : n
```

*Chaque générateur dispose de son compteur n **privé** !*

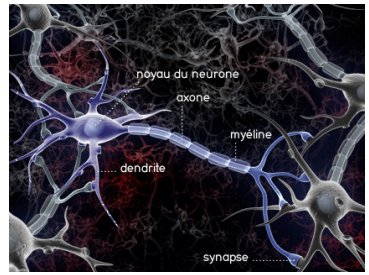
```
def make_gpair() :     # Python 3
    n = -2
    def resultat() :
        nonlocal n     # <-- !
        n = n + 2
        return n
    return resultat
```

```
>>> fermat = make_gpair()
>>> for i in range(5) :
      print(fermat(),end=' ')
0 2 4 6 8
>>> n
NameError: name 'n' is not defined
```


Les mémo-fonctions

• Une fonction définie en Scheme [une **fermeture**] dispose ainsi d'une **mémoire privée** !

• Imaginons cette mémoire privée comme constituée de **neurones** conservant de l'information. Utilisons-les pour **ne pas faire plusieurs fois les mêmes calculs** ! Cette idée, classique en I.A. est devenue une stratégie algorithmique [la ***programmation dynamique***]...

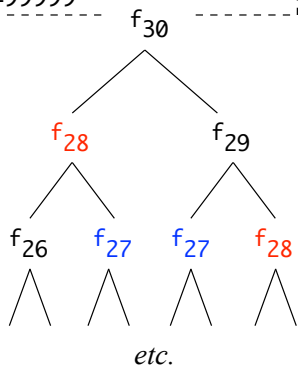


• Programmer des fonctions qui se souviennent des calculs qu'elles ont déjà effectués ! Des **mémo-fonctions**, ou **fonctions à mémoire**...

• La **fonction de Fibonacci** $f_0 = 0$, $f_1 = 1$, $f_n = f_{n-1} + f_{n-2}$ a une complexité exponentielle, car elle passe son temps à ***faire et refaire les mêmes calculs !***

```
(define (fib n)
  (when $trace? (set! cpt (+ cpt 1)))   ; juste pour voir...
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

```
> (define $trace? #t)
> (define cpt 0)
> (time (fib 30))
cpu time: 1665
832040
> cpt
2692537    ; !!!!
```



• ***Intelligence Artificielle*** : si j'avais un cahier de brouillon, j'y consignerais au fur et à mesure les calculs que je fais et leurs résultats, afin de ne pas les refaire !

• En Scheme : compiler la fonction `fib` dans un environnement privé pour ***stocker les calculs déjà faits dans une A-liste***.

• Pour exprimer que `fib(8) = 13` a déjà été calculé, on stockera le ***neurone*** `(8 13)` dans la A-liste `AL = (..... (8 13) .....)` représentant la mémoire privée d'une fermeture.

pour calculer intelligemment fib(n) :

- si fib(n) est déjà dans un neurone (n v), le résultat est v.

- sinon :

i) calculer récursivement fib(n) = fib(n-1) + fib(n-2)

ii) stocker fib(n) = v dans un nouveau neurone (n v)

iii) rendre v en résultat

• N.B. La mémoire grossit au fur et à mesure du calcul...

• L'implémentation de cette idée est précise mais pas difficile :

```
(define memo-fib
  (let (AL '((0 0) (1 1)))          ; les deux neurones initiaux
    (define (calcul n)              ; la procédure de calcul intelligente
      (define essai (assoc n AL))   ; calcul déjà fait ?
      (if essai
          (cadr essai)
          (let ((v (+ (calcul (- n 1)) (calcul (- n 2)))))
             (set! AL (cons (list n v) AL))     ; mémorisation !
             v)))
    calcul))                        ; le résultat est une fonction
```

• On accepte de **payer de l'espace mémoire pour gagner du temps** !

```
> (time (void (memo-fib 50000)))      ; complexité O(n)… Construction de la A-liste
cpu time: 178                         ; 178 ms
> (time (void (memo-fib 5000)))       ; il est déjà dans un neurone mais en profondeur…
cpu time: 3                           ; 3 ms
> (time (void (memo-fib 50000)))      ; il est déjà dans un neurone, mais c'est
cpu time: 0                           ; le dernier calculé donc il est en tête…
```

NB. Python utilise des *dictionnaires* (les *hash-tables* de Racket…).

• Reprogrammation de `memo-fib` (p. 20) en **remplaçant les A-listes par des hash-tables mutables** :

```
(define h-memo-fib
  (let ((H (make-hash '((0 . 0) (1 . 1)))))    ; ((n . v) ...)
    (define (calcul n)
      (if (hash-has-key? H n)
          (hash-ref H n)
          (let ((v (+ (calcul (- n 1)) (calcul (- n 2)))))
            (hash-set! H n v)                  ; mutation !
            v)))
    calcul))
```

```
> (time (void (memo-fib 50000)))      ; complexité O(n)… Construction de la hash-table
cpu time: 236                         ; 236 ms
> (time (void (memo-fib 5000)))       ; il est déjà dans un neurone, accès O(1)
cpu time: 0                           ; 0 ms
> (time (void (memo-fib 50000)))      ; il est déjà dans un neurone, accès O(1)
cpu time: 0                           ; 0 ms
```

• La construction d'une hash-table est *parfois* plus lente que celle d'une A-liste, mais l'accès est dans tous les cas beaucoup plus rapide !

Mieux que les A-listes : les tables de hash-code !

• Ce sont les **dictionnaires** de Python… TRES EFFICACES pour stocker des couples <variable>/<valeur>, accès quasiment O(1).

Python 3

```
>>> h = dict()
>>> h['pomme'] = 3
>>> h['poire'] = 2
>>> h
{'poire':2,'pomme':3}
>>> h['pomme'] = 4
>>> h
{'poire':2,'pomme':4}
>>> h['pomme']
4
>>> for k in h :
        print(k,d[k])
poire 2
pomme 4
```

Racket

```
> (define h (make-hash))
> (hash-set! h 'pomme 3)
> (hash-set! h 'poire 2)
> h
#hash((poire . 2) (pomme . 3))
> (hash-set! h 'pomme 4)
> h
#hash((poire . 2) (pomme . 4))
> (hash-ref h 'pomme)
4
> (for ([(k v) (in-hash h)])
    (printf "~a ~a\n" k v))
pomme 4
poire 2
```


Les hash-tables fonctionnelles (non mutables)

• Souvent, la table de hash-code est construite une fois pour toutes et ne subit pas de mutation. Dans ce cas, on peut l'utiliser de manière **purement fonctionnelle**, y-compris en *"rajoutant"* de nouveaux éléments.

• Le constructeur est alors `hash` et non `make-hash`. Le *modificateur* `hash-set!` n'existe plus, il est remplacé par `hash-set` qui étend la hash-table en un temps O(1) tout comme `cons` étend une liste sans la modifier… On peut passer une hash-table en paramètre.

```
> (define h (hash 'x 1 'y 2 'z 3))
> (define h1 (hash-set h 't 4))        ; O(1)
> h1
#hash((x . 1) (y . 2) (z . 3) (t . 4))
> h
#hash((x . 1) (y . 2) (z . 3))
```

Les ensembles utilisent aussi du hash-code

• Comme en Python, Racket offre les ensembles (mutables ou non), qui sont des collections d'éléments distincts. Ils utilisent du hash-code pour un accès très rapide, voir la doc…

Python 3

```
>>> E = set()
>>> E.add(4) ; E.add(1)
>>> E
{1,4}
>>> F = set([3,1,4,5])
>>> F
{1,3,4,5}
>>> F & E
{1,4}
```

Racket (version non mutable)

```
> (define E (list->set '(4 1)))
> E
#<set: 1 4>
> (define F (list->set '(3 1 4 5)))
> F
#<set: 1 3 4 5>
> (set-intersect E F)
#<set: 1 4>
```


Be careful with mutations !