

# Évaluation

Le mécanisme de l'évaluation est au coeur du fonctionnement de tout système de calcul formel.

## I. Affectation

Rappelons la syntaxe générale :

```
variable := expression ;
```

Dans cette situation :

- *expression* est évaluée -- l'objet de cette section,
- puis son résultat est stocké dans *variable*.

Ainsi, une variable *non encore affectée* se comporte comme si son contenu était simplement son *nom*<sup>1</sup>.

On dispose des fonctions suivantes pour savoir si une variable est affectée ou non :

- **anames()** (fonction sans argument) donne la séquence des variables affectées ;
- **unames()** donne la séquence des variables non affectées ;
- **assigned(variable)** renvoie *true* ou *false* selon l'état d'affectation de la *variable* ;
- **unassign(' variable ')** efface la *variable* (voir plus loin la nécessité des apostrophes) ;
- **restart** efface *toutes* les variables (de l'utilisateur).

```
/home/daniel/Documents/Maths/Présentations/unames - anames.mws
> x := 2 ; y ;
x := 2
y
> assigned(x) ; assigned(y) ;
true
false
> anames() ;
x
> unames() ;
identical, anyfunc, equation, positive, Integer, restart,
radical, And, host_based, gamma, neg_infinity, notice,
default, nomosint, nostnlot, days, left, relation,
invocations_left, specfunc, ansi, Catalan, posint,
warnlevel, Or, display, v, x, range, zppoly, gamma,
INTERFACE_SET, even, all, INTERVAL, false,
INTERFACE_HELD, labelling, atomic, X, numeric
```

Comme on le voit, **unames()** retourne un grand nombre de constantes et variables systèmes, en plus de celles (non affectées) de l'utilisateur.

## II. Évaluation

L'évaluation, c'est le "calcul" d'une quantité algébrique. Essentiellement, il s'agit de remplacer par leur contenu les variables qui en ont un.

<sup>1</sup> L'évaluation de cette variable ne produit rien d'autre que l'affichage de son nom.

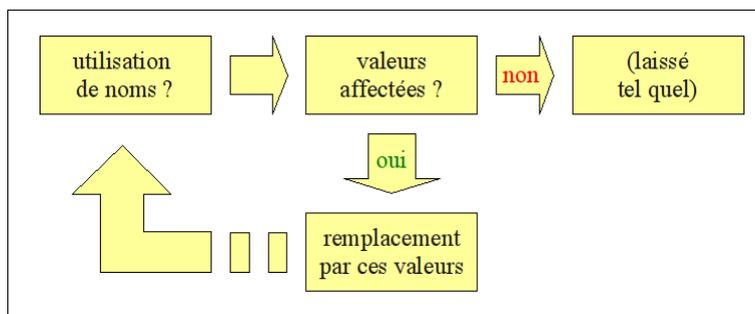
L'évaluation peut être

- *complète* ou *partielle*,
- *immédiate* ou *différée*.

Le cas général est celui d'une évaluation **complète et immédiate**.

### (a) Cas général

Voici ce qu'il advient d'une expression dans l'évaluation complète et immédiate :



Tant qu'il subsiste des noms de variables ayant un contenu ("*complète*"), le remplacement de ces variables par leurs contenus est effectué dans délai ("*immédiate*").

On comprend alors que la commande  $x := x+1$  avec  $x$  non affecté pose problème. On entrerait dans une succession infinie de remplacements ( $x+1$ ,  $x+2$ ,  $x+3$ ...) si *Maple* n'était pas pourvu d'un mécanisme de protection.

Examinons l'exemple suivant :

```
home/daniel/Documents/Maths/Présentations/Evaluation complète.mws
> x:=y : y:=z : z:=1 : x ;
1
```

À l'issue des trois affectations, les variables ont les contenus suivants :

variable	contenu
$x$	$y$
$y$	$z$
$z$	1

L'évaluation de  $x$  déclenche donc trois tours de la boucle décrite précédemment, où  $x$  se verra remplacé par  $y$ , puis  $z$ , puis finalement 1.

L'instruction `eval(x)` aurait eu le même effet dans ce cas<sup>2</sup>.

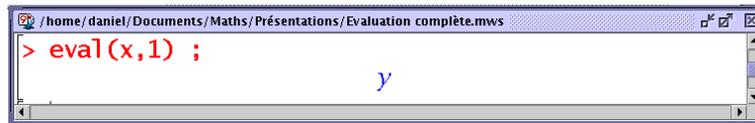
L'instruction `eval(x, n)` permet de forcer l'évaluation à un niveau  $n$  donné<sup>3</sup>.

En poursuivant l'exemple précédent, elle nous permet de constater que le contenu de  $x$  est bien  $y$  (et

<sup>2</sup> Cette instruction existe pour forcer l'évaluation complète dans les cas exceptionnels.

<sup>3</sup> Elle limite à  $n$  le nombre de tours de la boucle d'évaluation.

non 1).



```
/home/daniel/Documents/Maths/Présentations/Evaluation complète.mws  
> eval(x,1) ;  
y
```

Des exemples similaires, mais dans d'autres contextes, vont donner des résultats très différents :

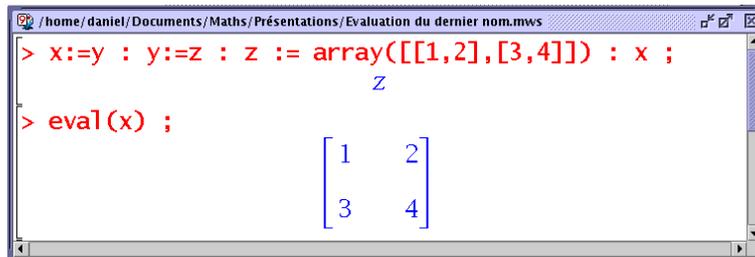
### (b) Structures "volumineuses"

La première exception est constituée par les "grosses" structures de données. Il s'agit des types *array*, *table*, *matrix*, *proc*.

*La règle dans ce cas est l'évaluation du dernier nom.*

Cela signifie que lorsque, dans la boucle d'évaluation, *Maple* rencontre un nom de variable dont le contenu est d'un des types précédents, ce nom *n'est pas* remplacé par son contenu.

Considérons la situation suivante, qui ne diffère de l'exemple initial que par le type du contenu de **z** :

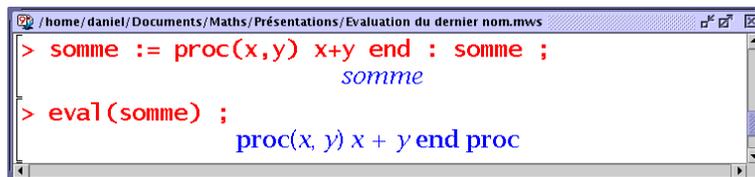


```
/home/daniel/Documents/Maths/Présentations/Evaluation du dernier nom.mws  
> x:=y : y:=z : z := array([[1,2],[3,4]]) : x ;  
z  
> eval(x) ;  

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```

Un autre exemple, utilisant une procédure :



```
/home/daniel/Documents/Maths/Présentations/Evaluation du dernier nom.mws  
> somme := proc(x,y) x+y end : somme ;  
somme  
> eval(somme) ;  
proc(x, y) x + y end proc
```

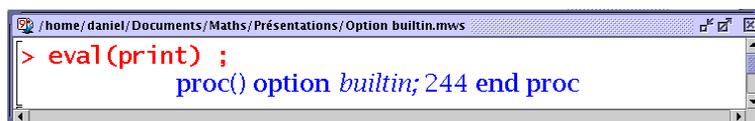
Il faut, dans ces cas, forcer l'évaluation complète avec **eval**.

Toutefois, cela ne nous donne pas encore accès aux fonctions de bibliothèque et autres fonctions de *Maple* qui sont programmées en *Maple* même.

On peut malgré tout accéder au code de ces fonctions après avoir saisi l'incantation : **interface(verboseproc = 2)**.

On constate sur le schéma (p. suivante) que celle-ci rend *Maple* beaucoup plus disert.

Toutefois, ce réglage n'est d'aucune utilité dans le cas des fonctions incorporées de *Maple*. Celles-ci ne sont pas programmées en *Maple* ; elles utilisent l'option **builtin** et résistent à tout examen :



```
/home/daniel/Documents/Maths/Présentations/Option builtin.mws  
> eval(print) ;  
proc() option builtin; 244 end proc
```

```

/home/daniel/Documents/Maths/Présentations/Fonctions de bibliothèques.mws
> eval(arcsin) ;
proc(x:algebraic) ... end proc
> interface(verboseproc = 2) ;
> eval(arcsin) ;
proc(x:algebraic)
local x1, s, k;
option system,
`Copyright (c) 1992 by the University of Waterloo. All rights reserved.`;
try return _Remember('procname'(args)) catch: end try
if nargs <> 1 then
error `expecting 1 argument, got %1`, nargs
elif type(x, 'complex(float)') then
evalf(arcsin(x))
elif type(x, 'infinity') then
if type(x, '{undefined, cx_infinity}') then
infinity + infinity*I
elif type(x, 'real_infinity') then
NumericEvent('real_to_complex', CopySign(1, x)^(1/2*Pi - infinity*I))
elif type(Re(x), 'infinity') then
1/2*CopySign(1, Re(x))*Pi + I*CopySign(infinity, Im(x))
else

```

### (c) Variable locales

Une autre exception survient à l'intérieur des procédures pour les variables locales. Celles-ci sont discutées dans le poly. "Fonctions et procédures".

Disons seulement que ce sont des variables qui n'existent que le temps qu'une procédure soit active. Donnons-en un exemple :

```

/home/daniel/Documents/Maths/Présentations/Variable locale.mws
> x:=1 :
> bidon:=proc() x:=2 end : # x est locale
> bidon() :
> x ;
1

```

Notons que *Maple* se permet de localiser automatiquement certaines variables dans les procédures :

- celles qui apparaissent au membre de gauche d'une affectation : **x := ...** ;
- les compteurs des boucles **for**.

La règle pour les variables locales est l'évaluation à un niveau.

On effectue donc un seul tour de la boucle d'évaluation.

Considérons l'exemple suivant, identique à celui du début, hormis qu'il est situé dans une procédure :

```

/home/daniel/Documents/Maths/Présentations/Evaluation à un niveau.mws
> test := proc() x:=y : y:=z : z:=1 : x end ;
test := proc() local x, y, z; x:= y; y:= z; z:= 1; x end proc
> test();
y

```

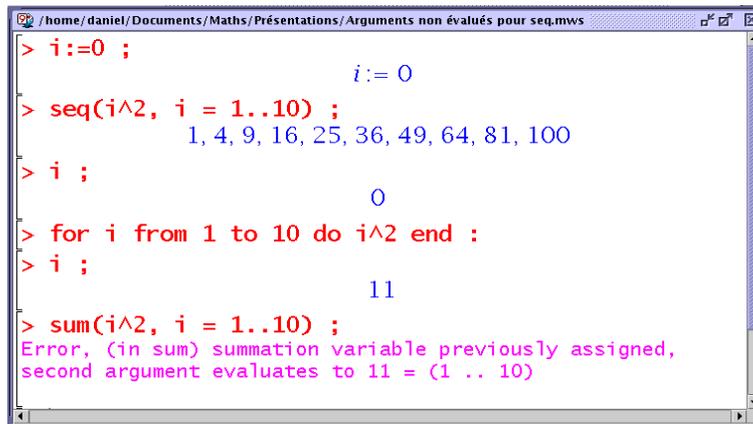
La variable **x**, locale à la procédure **test**, n'a été évaluée qu'au niveau 1.

Il faudra donc, dans les procédures, utiliser **eval** pour forcer l'évaluation complète si nécessaire.

#### (d) Commandes **seq** et **for**

**seq** n'évalue pas ses arguments, et ne les affecte pas non plus. **for** les affecte sans les évaluer.

Cela peut produire des résultats inattendus :



```
> i:=0 ;
                                     i := 0
> seq(i^2, i = 1..10) ;
                                     1, 4, 9, 16, 25, 36, 49, 64, 81, 100
> i ;
                                     0
> for i from 1 to 10 do i^2 end ;
> i ;
                                     11
> sum(i^2, i = 1..10) ;
Error, (in sum) summation variable previously assigned,
second argument evaluates to 11 = (1 .. 10)
```

Par comparaison, **sum** et **prod** affectent initialement la variable (et l'évaluent). Si celle-ci est déjà affectée, cela produit une erreur.

Il est donc parfois utile de différer l'évaluation.

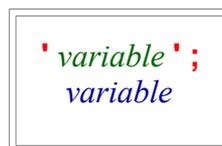
#### (e) Évaluation différée

On diffère l'évaluation d'une variable à l'aide de guillemets simples (apostrophes).

On fera attention à ne pas confondre :

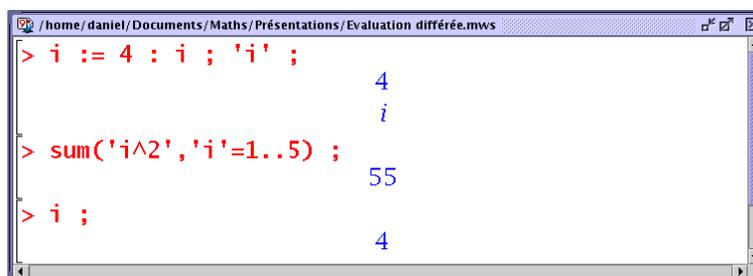
- `'` (sous 4) "right quote" : c'est lui qui sert à différer l'évaluation ;
- ``` (Alt Gr + 7) "left quote" ou "backquote" : délimiteur de chaînes de caractères.

La manière générale de différer l'évaluation d'une variable est donc :



```
'variable';
variable
```

Voici un exemple :



```
> i := 4 : i ; 'i' ;
                                     4
                                     i
> sum('i^2', 'i'=1..5) ;
                                     55
> i ;
                                     4
```

L'évaluation complète retire un niveau de guillemets simples `'`. Ceux-ci ne font que *différer* l'évaluation de la variable.

D'autre part, *ils n'empêchent pas* les simplifications automatiques.

Une application possible est l'effacement d'une seule variable **x** :

- **unassign(' x ')** ;
- ou tout simplement **x := ' x '** .

Dans les deux cas, les guillemets simples sont bien sûr indispensables<sup>4</sup> .

Dans le cas des tableaux p. ex. :

```

/home/daniel/Documents/Maths/Présentations/Evaluation différée des tableaux.mws
> a := array([1,2,3,4]) : i := 3 : 'a[i]' ; % ;
      a_i
      3
>

```

on peut remarquer que les guillemets simples "passent à travers" les crochets et diffèrent aussi l'évaluation de l'indice.

### (f) Paramètres (arguments) des procédures

Un simple mot pour mentionner que *ce ne sont pas des variables*.

Ce sont des *valeurs* qui sont récupérées au moment de l'appel de la procédure. Il n'y a *pas* de création de variable homonyme<sup>5</sup>.

Plus de détails dans le poly. "Fonctions et procédures", notamment sur les modalités particulières d'évaluation de ces paramètres.

## III. Substitutions

Les remarques sur l'évaluation éclairent le mécanisme des substitutions, quand plusieurs ont lieu dans une même instruction **subs** :

- elles sont généralement *successives* (il y a *évaluation* après chacune d'elles) ;
- sauf si on les regroupe avec des accolades :

```

/home/daniel/Documents/Maths/Présentations/Substitutions.mws
> subs( x=y , y=z , z=x , x+2*y^2+3*z^3) ;
      x + 2 x^2 + 3 x^3
> subs({x=y , y=z , z=x}, x+2*y^2+3*z^3) ;
      y + 2 z^2 + 3 x^3

```

Dans ce second cas, on voit qu'il n'y a *pas d'évaluation intermédiaire* (entre les **{ }**), et que les substitutions sont réellement *simultanées*.

4 Sans eux, **x** serait remplacé par son contenu.

5 On parle de "*transmission par valeur*" des arguments.