

Notes de cours Maple

Claude Gomez

INRIA
Domaine de Voluceau
Rocquencourt - BP105
78153 Le Chesnay Cédex
France
email: Claude.Gomez@inria.fr

Janvier 2004

1 Introduction

Maple a été conçu vers 1980 par des chercheurs de l'université de Waterloo. Il est aujourd'hui commercialisé par Waterloo Maple Incorporated.

Il fonctionne sur les stations de travail UNIX et GNU/Linux, sur Macintosh et sous Windows. Ces notes de cours sont valables pour les versions 6, 7 et 8 de Maple.

2 Déroulement d'une session

L'appel de Maple est différent selon les machines utilisées.

Il existe maintenant deux modes pour entrer les données : l'ancien mode « Maple Notation » et le mode « Standard Math » qui utilise les palettes de formules. C'est l'ancien mode que nous utiliserons.

Dans ce mode :

- le prompt est : >
- chaque commande doit se terminer par ; (résultat affiché) ou par : (résultat non affiché)
- pour sortir de Maple : `quit` (← mot-clé) ou on utilise un menu.

3 Commandes utiles

- `help : ?< sujet >`
- Réinitialisation de la session Maple : utiliser `restart`.
- `%` a pour valeur le dernier résultat, `%%` l'avant-dernier, et `%%%` l'antépénultième.
- Lors de son lancement, Maple charge le fichier d'initialisation de l'utilisateur. Son nom et son emplacement dépendent du système utilisé. Par exemple, sur une station de travail UNIX, ce fichier a pour nom `.mapleinit` et doit se trouver dans le home directory de l'utilisateur. Sous Windows 9X/2000/XP ce fichier est appelé `maple.ini` et doit se trouver dans le directory courant de l'utilisateur.

- Temps d'exécution : la fonction `time()` donne le temps CPU en secondes depuis le début de la session.
Une façon d'avoir le temps d'un calcul :
`> st :=time() :<calcul> :time()-st;`
Il existe d'autres moyens : fonctions `showtime`, `profile`.
`time(<expression>)` donne le temps de calcul de l'expression `<expression>`.
- La fonction `interface` permet de modifier l'interface utilisateur avec Maple.
- `print(<suite d'expressions>)` permet de faire une impression en 2 dimensions. Il existe aussi la fonction `printf` qui ressemble à celle du langage C.
- `lprint(<suite d'expressions>)` permet de faire une impression linéaire.
- Pour lire des valeurs à la console, utiliser la fonction `readstat`.

4 Expressions

4.1 Constantes

- Nombres : entiers, rationnels, flottants (nombre de chiffres significatifs = valeur de `Digits`, défaut 10) qui peuvent être écrits :
`Float(<mantisse>, <exposant>)`, par exemple `Float(12.5,4)` ou bien de façon habituelle `12.5e4`
- Constantes : `false`, `true`, `infinity`, `Pi`...
Toutes les constantes sont dans la variable `constants`. On ne peut pas modifier leur valeur.
La constante e (base des logarithmes népériens) est représentée par `exp(1)` et le i des complexes par `I`.

4.2 Chaînes de caractères

Une chaîne de caractères (type `string`) se définit en l'encadrant de doubles quotes " :
`"Je suis une chaîne de caractères"`.

On ne peut pas attribuer de valeur à une chaîne de caractères.

On utilise l'opérateur de sélection `[]` pour accéder aux caractères ou aux sous-chaînes de la chaîne de caractères.

4.3 Noms

Un nom (type `name`) a une valeur qui peut être lui-même ou n'importe quelle expression. Un nom est un symbole (type `symbol`), mais ce peut être aussi un nom indexé (type `indexed`) (voir 12).

Pour former un nom :

- `toto`, `tata`... mais on ne peut pas utiliser des mots-clés (30 mots-clés). Pour les connaître faire `?keywords`
- '`<n'importe quoi>`' ← ce sont des backquote
par exemple `'n-1' := 12;` est correct
Si l'on veut mettre un backquote, on le double.

Il n'y a pas de différence entre `toto` et `'toto'`.

Maple fait la différence entre les majuscules et les minuscules.

Pour assigner une valeur à un nom, on utilise `:=`

Si dans l'assignation on veut évaluer la variable de gauche, on utilise la fonction `assign`. Par exemple :

```
> a:=b:
> assign(a,2):
> b;
```

2

Pour créer des noms, on peut utiliser l'opérateur de concaténation « || ». Tous les opérandes sont évalués sauf le premier :

```
> a||1;
```

a1

```
> a||b||toto;
```

abtoto

```
> a:=b:
```

```
> a||c;
```

ac

Si l'on veut évaluer le premier opérande, on met au début un nom vide :

```
> a:=b:
```

```
> ""||a||c;
```

bc

En Maple, on peut en général utiliser un nom à la place d'une chaîne de caractères. En revanche la chaîne de caractères a pour valeur elle-même et on ne peut pas lui assigner de valeur.

4.4 Opérateurs

Opérateurs classiques : + * - / ^

Fonctions mathématiques résidentes dans Maple (faire `?inifcns` pour en avoir la liste) : `sin`, `cos`...

Opérateurs logiques : < <= > >= <> and or not

4.5 Intervalle (range)

`<expression>..<expression>` : exprime une variation entre les deux expressions.

On en verra plus tard l'utilisation pratique.

4.6 Suite d'expressions (sequence ou expression sequence)

– C'est une expression de la forme :

`<expr1>, <expr2>, ..., <exprn>`

NULL est la suite ne contenant pas d'élément.

On peut assigner une suite d'expressions à un nom :

```
a:=b,c,d;
```

– On peut former des suites d'expressions à partir de suites d'expressions :

```
> l:=b,c:
```

```
> m:=a,l,d;
```

m := a, b, c, d

- Utilisation de l'intervalle :

```
> p|| (1..5);
```

$p1, p2, p3, p4, p5$

Avec une fonction :

```
> f(p|| (1..2));
```

$f(p1, p2)$

```
> a:=a1,a2,a3:
```

```
> f(a);
```

$f(a1, a2, a3)$

- L'opérateur de répétition (« sequence operator ») \$ permet de construire des suites d'expressions :

```
<expr1>$<nom>=<expr2>..<expr>
```

produit une suite d'expressions avec chaque $\langle expr1 \rangle$ remplacé par sa valeur lorsque $\langle nom \rangle$ décrit l'intervalle $\langle expr2 \rangle .. \langle expr3 \rangle$

Par exemple :

```
> i^2$i=1..10;
```

1, 4, 9, 16, 25, 36, 49, 64, 81, 100

Attention, l'expression à gauche du \$ est évaluée d'abord :

```
> a||i$i=1..10;
```

$ai, ai, ai, ai, ai, ai, ai, ai, ai, ai$

```
> 'a||i'$i=1..10;
```

$a1, a2, a3, a4, a5, a6, a7, a8, a9, a10$

Formes simplifiées :

$\langle expr1 \rangle \$ \langle expr3 \rangle$ est équivalent à $\langle expr1 \rangle \$ i = 1 .. \langle expr3 \rangle$

```
> x$4;
```

x, x, x, x

$\$ \langle expr2 \rangle .. \langle expr3 \rangle$ est équivalent à $i \$ i = \langle expr2 \rangle .. \langle expr3 \rangle$

```
> $1..5;
```

1, 2, 3, 4, 5

On peut souvent (et avantageusement) remplacer cet opérateur par la fonction `seq` (voir 11.2.1).

- Pour récupérer un élément d'une suite d'expressions, on utilise l'opérateur de sélection $[\dots]$:

```
> a:=b,c,d:
```

```
> a[2];
```

c

4.7 Ensembles, listes

Un ensemble (type `set`) est non ordonné : $\{\langle suite d'expressions \rangle\}$

Une liste (type `list`) est ordonnée : $[\langle suite d'expressions \rangle]$

$\{ \}$ est l'ensemble vide et $[]$ est la liste vide.

Le tableau 1 résume les principales différences entre une suite d'expressions, une liste et un ensemble.

TAB. 1 – Listes, ensembles et suites d'expressions

Objet	Caractéristiques
liste	type : list syntaxe : $[a, b, c]$ ordonné : $[a, b, c] \neq [b, a, c]$ éléments répétés : $[a, a, c] \mapsto [a, a, c]$ plusieurs niveaux : $l := [b, c]$; $[a, l, d] \mapsto [a, [b, c], d]$ liste vide : $[]$
ensemble	type : set syntaxe : $\{a, b, c\}$ non ordonné : $\{a, b, c\} = \{b, a, c\}$ pas d'éléments répétés : $\{a, a, c\} \mapsto \{a, c\}$ plusieurs niveaux : $l := \{b, c\}$; $\{a, l, d\} \mapsto \{a, \{b, c\}, d\}$ ensemble vide : $\{\}$
suite d'expressions	type : exprseq syntaxe : a, b, c ordonné : $a, b, c \neq b, a, c$ éléments répétés : $a, a, c \mapsto a, a, c$ un seul niveau : $l := b, c$; $a, l, d \mapsto a, b, c, d$ suite vide : NULL

- L'opérateur de sélection permet de prendre un élément ou une suite d'éléments. Le problème est que pour les ensembles on ne connaît pas l'ordre a priori. Par exemple pour des listes :

```

> l := [1, 2, 3] :
> l[2] ;
                2
> l[1..2] ;
                [1, 2]
> s := 1, 2, 3 :
> l := [s, 4] ;
                l := [1, 2, 3, 4]
> l[] ;
                1, 2, 3, 4

```

On peut aussi utiliser l'opérateur de sélection pour changer l'élément d'une liste :

```

> l := [1, 2, 3] :
> l[2] := 20 ;
                l2 := 20
> l ;
                [1, 20, 3]

```

Pour avoir le nombre d'éléments d'un ensemble ou d'une liste, il faut utiliser la fonction `nops` (voir 5).

- En fait, les listes se comportent comme des vecteurs lignes et si l'on soustrait deux listes de même taille, le résultat est une liste dont tous les éléments sont des 0. Si elles ne sont pas de même taille, on obtient une erreur et il vaut mieux utiliser l'opérateur de relation = avec `evalb` pour savoir si elles sont égales :

```
> l1:=[1,2,3]:
> l2:=[1,2,3]:
> l1-l2;
                                [0, 0, 0]

> l3:=[1,2,3,4]:
> evalb(l1=l3);
                                false
```

- Pour les ensembles on a les opérateurs suivants :
`union minus intersect`
- Pour savoir si des ensembles sont égaux il faut utiliser l'opérateur de relation = avec `evalb` car on ne peut pas les soustraire.
- Fonctions utiles pour les listes et les ensembles : `member select map zip` (voir 11.2.3).

5 Les fonctions `op`, `nops` et `subsop`

- `op` est une fonction très puissante de Maple qui extrait les opérandes d'une expression :
`op(<i>, <expr>)` extrait l'opérande numéro <i> de <expr>
`op(<i>..<j>, <expr>)` extrait la suite d'expressions des opérandes de <i> à <j>
`op(<expr>)` extrait la suite d'expressions de tous les opérandes.
- `nops(<expr>)` donne le nombre d'opérandes.
- `subsop(<i>=<expri>, <expr>)` permet de remplacer l'opérande numéro <i> de <expr> par <expri>.

```
> e:=a+b*c^2+d:
> op(e);
                                a, b c^2, d

> op(1,e);
                                a

> op(1..2,e);
                                a, b c^2

> nops(e);
                                3

> subsop(2=x,e);
                                a + x + d
```

- En particulier `op` peut s'utiliser à la place de l'opérateur de sélection pour les listes, les ensembles et les suites d'expression. Cela permet de remplacer la plupart des fonctions classiques de manipulations de listes.

```
> l:=[1,2,3]:
> op(l);
                                1, 2, 3
```

- ```

> nops(1);
3
> subsop(2=x,1);
[1, x, 3]

```
- subsop permet donc aussi de changer un élément d'une liste.
- Parfois `op(0, <>)` a un sens. Si ni `g` ni `a` n'ont de valeur :

```

> op(0,g(a,b));
g
> op(0,a[b,c]);
a

```
  - Pour un nom indexé (table ou tableau n'ayant pas de valeur), `op` donne la suite des indices « les plus à droite » :

```

> op(t[1,2,3][4,5]);
4, 5
> op(0,t[1,2,3][4,5]);
t1,2,3
> op(1,t[1,2,3][4,5]);
4

```
- On verra l'utilisation plus poussée de `op` pour les tables (voir 12).

## 6 L'évaluation dans Maple

Toute expression Maple syntaxiquement correcte retourne une valeur (sauf erreur d'exécution). Cette valeur peut être la suite d'expressions nulle `NULL`.

### 6.1 Les noms

- Les nombres, les chaînes de caractères et les noms sans valeur assignée sont auto-évaluables :

```

> toto;
toto
> a[4];
a4
> 10;
10

```

Rappel : nom = symbole ou nom indexé.

- Pour assigner une valeur à un nom, on utilise :
  - `<nom> := <valeur>` où `<nom>` n'est pas évalué ou bien
  - `assign(<nom>, <valeur>)` où `<nom>` est évalué.`<nom>` ne peut pas être un mot-clé.
- Que vaut `<nom>` ?
  - Mécanisme de « full evaluation » : une fois qu'une valeur est assignée à une variable, elle prend cette valeur partout où elle apparaissait avant :
    - l'assignation rajoute un pointeur de la variable vers la valeur,
    - pour obtenir la valeur de la variable on « suit » les pointeurs jusqu'au bout.

Si l'on veut supprimer l'évaluation, il faut « quoter » : 'a'

```
> c:=b: # c -> b
> b:=a: # c -> b -> a
> a:=1: # c -> b -> a -> 1
> b;
1
> c;
1
> b:='b': # c -> b et a -> 1
> c;
b
> b;
b
> a;
1
```

`b :='b'` est la « désassignation » de `b`.

En revanche `b :='b'` (ce sont des backquotes) est strictement équivalent à `b :=b`.

```
> c:=b: b:=a: a:=1: # c -> b -> a ->1
> b:='b': # c -> b -> 1 et a -> 1
> c;
1
> b;
1
> a;
1
```

– Si l'on veut savoir vers quoi pointe une variable, on l'évalue à un seul niveau `eval(<nom>,1)` :

```
> c:=b: b:=1: # c -> b -> 1
> eval(b,1);
1
> eval(c,1);
b
```

– Si l'on veut savoir si une valeur a été assignée à un nom, on utilise `assigned(<nom>)` où `<nom>` n'est pas évalué :

```
> a:=b:
> assigned(a);
true
> assigned(b);
false
```

– `anames()` ; retourne la suite d'expressions des variables auxquelles une valeur a été assignée.

– Attention à l'erreur :

```
> b:='a': # b --> a
> a:='b': # a --> b
> a; # evaluation infinie !!!
```

on n'en revient plus!

- A l'intérieur des procédures, le mécanisme d'évaluation n'est plus le même (voir section 13).

## 6.2 Les opérateurs

Tous les opérandes sont évalués.

## 6.3 Les expressions fonctionnelles

Une expression fonctionnelle est de la forme  $\langle nom \rangle (\langle arg1 \rangle, \dots, \langle argn \rangle)$

- Tous les arguments sont évalués de gauche à droite, ensuite  $\langle nom \rangle$  est évalué :
  - si c'est une procédure, elle est exécutée sur les arguments évalués
  - si aucune valeur n'est assignée à  $\langle nom \rangle$ , Maple retourne l'application de la fonction aux arguments évalués (type **function**)
  - si  $\langle nom \rangle$  a une valeur qui n'est pas une procédure, ce n'est pas une erreur :

```
> a:=1:
```

```
> a(f);
```

1

```
> a:=b+c:
```

```
> a(1);
```

b(1) + c(1)

- Lorsque l'on veut assigner comme valeur une procédure à un nom, on utilise l'assignation  $\langle nom \rangle := \text{proc} \langle \rangle \text{end};$

Ici le mécanisme de « full evaluation » est limité par le mécanisme de « last name evaluation ». L'évaluation s'arrête au dernier nom « avant » la procédure.

Si une procédure a été assignée comme valeur à **f**, l'évaluation du nom **f** va donner **f**. C'est **eval(f)** qui retournera l'objet procédure. La fonction **print** permet d'imprimer la procédure.

On a donc les résultats suivants :

```
> f:=proc(x) 1+x; end: # f -> procedure
```

```
> f;
```

*f*

```
> eval(f);
```

**proc(x) 1 + x end proc**

```
> print(f);
```

**proc(x) 1 + x end proc**

```
> f(2);
```

3

```
> g:=f: # g -> f -> procedure
```

```
> g(2);
```

3

```
> g;
```

*f*

```

> eval(g);
 proc(x) 1 + x end proc
> f:=1; # g -> f -> 1
 f := 1
> g(2);
 1

```

Noter la différence avec :

```

> f:=proc(x) 1+x; end; # f -> procedure
> g:=eval(f): # f -> procedure <- g
> g;
 g
> f:=1: # f -> 1 et procedure <- g
> g(2);
 3

```

- Très peu d'exceptions à ce mode d'évaluation des expressions fonctionnelles. Seules quelques fonctions n'évaluent pas certains de leurs arguments. C'est le cas par exemple de **assigned**, **eval** et **evaln** (voir 6.5), **evalhf**, **seq** (voir 11.2.1) et **userinfo**.
- Certaines fonctions Maple fournies par le système peuvent avoir une forme inerte. C'est le même nom de fonction avec la première lettre en majuscule. On les appelle fonctions inertes (« inert functions »).

Ces fonctions ne sont pas évaluées et servent d'abord comme « place » pour exprimer quelque chose :

- si l'on ne veut pas évaluer la fonction :

```

> Int(x^2,x);
 $\int x^2 dx$
> Factor(x^2);
 Factor(x^2)

```

- si l'on ne sait pas évaluer la fonction. Remarquer la différence avec le résultat donné avec la fonction non inerte. Les comportements peuvent être différents :

```

> factor(x^2+3*x+3);
 $x^2 + 3x + 3$
> Factor(x^2+3*x+3);
 Factor(x^2 + 3x + 3)

```

Ces fonction inertes sont connues par un certain nombres de fonctions et d'opérateurs Maple et sont alors évaluées par ces derniers. Deux grands groupes :

- celles connues par **mod**, **evala**, **modp1** ou **evalgf** :

```

> Factor(x^2+3*x+3) mod 7;
 (x + 4)(x + 6)

```

- celles connues par **evalf** permettant de réaliser des calculs numériques :

```

> evalf(Int(exp(x^3),x=0..1));
 1.341904418

```

Enfin la fonction `value` permet de rendre active la fonction inerte en l'évaluant :

```
> i:=Int(1/(1+x^3),x);
```

$$i := \int \frac{1}{1+x^3} dx$$

```
> value(i);
```

$$-\frac{1}{6} \ln(x^2 - x + 1) + \frac{1}{3} \sqrt{3} \arctan\left(\frac{(2x-1)\sqrt{3}}{3}\right) + \frac{1}{3} \ln(x+1)$$

## 6.4 Les tables et les tableaux

Le fonctionnement est équivalent à celui des procédures. On a aussi le mécanisme de « last name evaluation » (voir 6.3).

```
> t:=table([1=foo,2=bar]): # t -> table
> t;
t
> eval(t);
table([1 = foo, 2 = bar])
> print(t);
table([1 = foo, 2 = bar])
> t[1];
foo
> a:=t: # a -> t -> table
> a[1];
foo
> a;
t
> eval(a);
table([1 = foo, 2 = bar])
> t:=1; # a -> t -> 1
t := 1
> a[2];
1_2
```

Noter la différence avec :

```
> t:=table([1=foo,2=bar]): # t -> table
> a:=eval(t): # t -> table <- a
> a;
a
> t:=1: # t -> 1 et table <- a
> a[1];
foo
```

Il n'y a pas d'ambiguïté avec l'opérateur de sélection `[...]` appliqué à un nom car un nom ne peut être à la fois une table, une liste, un ensemble ou une suite d'expressions.

## 6.5 Les évaluateurs de Maple

- La fonction `eval(<nom>, <n>)` où `<nom>` n'est pas évalué permet de retourner l'évaluation de `<nom>` au niveau `<n>` (argument optionnel). Si ce dernier argument est omis, le niveau d'évaluation est « infini ».

```
> a:=b: b:=c: c:=1: # a -> b -> c -> 1
> eval(a);
1
> eval(a,2);
c
```

De façon pratique, on peut utiliser `eval` avec un deuxième argument pour évaluer une expression pour une valeur d'un nom :

```
> eval(sin(a), a=Pi);
0
```

Cette méthode évite d'assigner une valeur à ce nom, ce qui a un effet sur les autres expressions où intervenait ce nom.

- `evalb` évalue une expression booléenne.
- `evalf(<expr>, <n>)` permet d'évaluer numériquement l'expression `<expr>` avec des nombres flottants. Le nombre de chiffres des flottants utilisés est égal à `<n>` (argument optionnel). Si ce dernier argument est omis, le nombre de chiffres des flottants est égal à la valeur de la variable globale `Digits` (10 par défaut).

Attention, ce nombre de chiffres n'est pas le nombre de chiffres significatifs : lors des calculs, la précision peut diminuer.

On peut utiliser les flottants de la machine (double précision) pour aller plus vite : c'est la fonction `evalhf`.

- la fonction `evaln` évalue pour donner un nom :

```
> i:=1:
> evaln(i);
i
> 'a||i';
a||i
> evaln(a||i);
a1
> 'a[i]';
ai
> evaln(a[i]);
a1
```

- `evalm` permet d'évaluer les expressions matricielles. On verra cette fonction plus en détail dans le chapitre sur les tableaux (section 12.4.1).
- Il y a d'autres évaluateurs : `evala`, `evalb`, `evalc`...

## 7 La simplification dans Maple

- Il y a une table de simplification dans laquelle toutes les expressions et sous-expressions sont stockées. En conséquence chaque expression simplifiée a une unique instance en mémoire.

Chaque expression Maple est « simplifiée », puis comparée avec celles de la table. Si elle n'est pas dans la table, une nouvelle entrée est créée, sinon cette nouvelle expression est supprimée et c'est celle de la table qui est utilisée. Ce processus est réalisé à l'aide de hash tables.

Chaque expression simplifiée a une signature, et ce sont les signatures que l'on compare. Si elles sont égales, on fait une comparaison complète.

Ici, « simplifié » signifie « simplification de base » : addition, multiplication, puissance...

Tout ceci implique que **l'on ne doit pas** se baser sur l'ordre des opérandes des expressions que l'on entre :

```
> b+a+c;
```

$$b + a + c$$

```
> a+b+c;
```

$$b + a + c$$

Il n'y a pas de simplifications non demandées (pas de réduction au même dénominateur, de développement, de mise sous forme canonique automatique).

- Lorsqu'une expression est entrée dans Maple et simplifiée par le simplificateur, il remplace dans la représentation interne :

-a par  $(-1)*a$

1/a par  $a^{-1}$

Donc en représentation interne :

a-b est  $a+(-b)$

et a/b est  $a*b^{-1}$

Il faut donc faire attention lorsque l'on utilise `whattype`, `op` et `convert` :

```
> whattype(a-b);
```

$$+$$

```
> op(a/b);
```

$$a, \frac{1}{b}$$

```
> convert(a-b, '*');
```

$$-a b$$

- La fonction `normal` permet de transformer une expression en représentation normale d'expressions rationnelles. Elle est à utiliser en priorité avant d'essayer la fonction `simplify`.
- La fonction `simplify` est la fonction générale permettant de simplifier des expressions : `simplify(<expr>, <nom1>, ..., <nomn>)` où `<nom1>, ..., <nomn>` sont des noms optionnels représentant des règles de simplification.

Sans argument optionnel, `simplify` applique un ensemble de règles de simplification. S'il y a un ou des noms optionnels, il n'applique que les règles correspondantes. Quelques noms : `trig`, `ln`, `exp`, `sqrt`, `radical`...

**Attention**, il est parfois dangereux d'utiliser la fonction `simplify` sans argument optionnel ; en effet la simplification est alors réalisée sans contrôle de l'utilisateur ni connaissance par Maple des types mathématiques des données (entiers, réels ou complexes...). Cela peut alors donner des résultats faux dans le cas de fonctions multiformes (logarithmes complexes par exemple).

- On peut aussi utiliser des règles de réécriture :

```
simplify(<expr>, {a1=b1, ..., an=bn})
```

où `{a1=b1, ..., an=bn}` est un ensemble d'équations.

- La fonction `rationalize` simplifie des expressions ayant des racines carrées au dénominateur.
- La fonction `expand` développe une expression et peut donc simplifier.
- La fonction `combine` combine des sommes, produits, puissances de termes en un seul terme. Comme `simplify` elle prend un argument optionnel qui peut être `exp`, `ln`, `power` ou `trig`.
- Une autre fonction très utile qui est très proche de la simplification est la fonction `convert(<expr>, <forme>, <arg1>, ..., <argn>)` qui convertit une expression d'une forme vers une autre. `<arg1>, ..., <argn>` sont des arguments optionnels qui dépendent de `<forme>` (73 formes possibles!).

Cette fonction est très utile lorsque l'on veut faire des transformations :

```
> convert([a,b,c], '+');
 a + b + c
> convert([a,b,c], '*');
 a b c
> convert ([a,b,c], array);
 [a, b, c]
> convert(%, list);
 [a, b, c]
```

Il faut penser à ces fonctions lorsque l'on veut transformer des expressions. Le tableau 2 décrit des exemples d'utilisation de ces fonctions de simplification.

## 8 Modifier le comportement de Maple

On peut rajouter à un certain nombre de fonctions Maple des extensions permettant d'ajouter des simplifications à ces fonctions ou de nouvelles fonctionnalités. C'est ce que l'on appelle l'interface utilisateur (« user interface ») de la fonction. C'est en particulier le cas pour `convert`, `diff`, `evalf`, `expand`, `modp`, `series` et `simplify`.

On peut ainsi modifier le simplificateur de Maple. En fait, chaque appel `simplify(<expr>, <nom>)` est un appel à la fonction

```
'simplify/<nom>' (<expr>).
```

On peut donc écrire soi-même des fonctions de simplification supplémentaires qui seront alors connues de la fonction `simplify`. Il faut s'aider du source des fonctions existantes qui se trouve dans la librairie Maple (voir 16); ce n'est pas très difficile grâce à cela.

Il en est de même pour la fonction `convert`. On peut définir la procédure :

```
'convert/<f>'
et 'convert/<f>'(a,x,y,...) par exemple sera appelé lors de l'exécution de
convert(a,<f>,x,y,...).
```

La façon de modifier le comportement de ces fonctions est décrite dans le manuel de chaque fonction.

## 9 Les types

Maple reconnaît 110 types de base!

Exemple : `integer`, `fraction`, `float`, `string`, `name`, `'+'`, `'*'`, `'^'`, `equation`, `list`, `number`...  
Ce sont des types de base (types système). Faire `?type` pour les avoir tous.

TAB. 2 – Quelques simplifications de fonctions

| Commande                                                      | Résultat                                                                                                                                                                                                               |
|---------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>expand(...)</code>                                      | $\sin(a + b) \mapsto \sin(a) \cos(b) + \cos(a) \sin(b)$<br>$\cos(a + b) \mapsto \cos(a) \cos(b) - \sin(a) \sin(b)$<br>idem pour les fonctions hyperboliques<br>$e^{a+b} \mapsto e^a e^b$                               |
| <code>combine(..., trig)</code>                               | $\cos(a) \cos(b) \mapsto \cos(a - b)/2 + \cos(a + b)/2$<br>$\cos(a) \sin(b) \mapsto \sin(a + b)/2 - \sin(a - b)/2$<br>$\sin(a) \sin(b) \mapsto \cos(a - b)/2 - \cos(a + b)/2$<br>idem pour les fonctions hyperboliques |
| <code>combine(..., exp)</code>                                | $e^a e^b \mapsto e^{a+b}$<br>$e^{a+n \ln b} \mapsto b^n e^a$ où $n$ est entier                                                                                                                                         |
| <code>combine(..., ln)</code>                                 | $\ln a + \ln b \mapsto \ln(ab)$ si $a > 0$ et $b > 0$                                                                                                                                                                  |
| <code>simplify(..., trig)</code>                              | $\sin(x)^2 + \cos(x)^2 \mapsto 1$<br>$\cosh(x)^2 - \sinh(x)^2 \mapsto 1$                                                                                                                                               |
| <code>simplify(..., exp)</code>                               | $e^{a \ln b} \mapsto b^a$                                                                                                                                                                                              |
| <code>simplify(..., ln)</code><br>ou <code>expand(...)</code> | $\ln(b^a) \mapsto a \ln b$ si $a > 0$ et $b > 0$<br>$\ln(ab) \mapsto \ln a + \ln b$ si $a > 0$ et $b > 0$                                                                                                              |
| <code>simplify(..., power)</code>                             | $x^a x^b \mapsto x^{a+b}$                                                                                                                                                                                              |
| <code>simplify(..., power, symbolic)</code>                   | $(a^b)^c \mapsto a^{bc}$<br>$\sqrt{a^2} \mapsto a$                                                                                                                                                                     |
| <code>convert(..., exp)</code>                                | $\cos(x) \mapsto (e^{ix} + e^{-ix})/2$<br>$\cosh(x) \mapsto (e^x + e^{-x})/2$                                                                                                                                          |
| <code>convert(..., trig)</code>                               | $e^{ix} \mapsto \cos(x) + i \sin(x)$<br>$e^x \mapsto \cosh(x) + \sinh(x)$                                                                                                                                              |
| <code>convert(..., ln)</code>                                 | $\arccos(x) \mapsto -i \ln(x + i\sqrt{1 - x^2})$<br>$\operatorname{arctanh}(x) \mapsto (\ln(1 + x) - \ln(1 - x))/2$                                                                                                    |
| <code>convert(..., tan)</code>                                | $\sin(x) \mapsto \frac{2 \tan(x/2)}{1 + \tan^2(x/2)}$<br>$\cos(x) \mapsto \frac{1 - \tan^2(x/2)}{1 + \tan^2(x/2)}$<br>idem pour les fonctions hyperboliques                                                            |
| <code>convert(..., sincos)</code>                             | $\tan(x) \mapsto \frac{\sin(x)}{\cos(x)}$<br>idem pour les fonctions hyperboliques                                                                                                                                     |

A partir de ces types de base, on peut construire des types structurés : un type structuré est une expression Maple qui peut être interprétée comme un type. Pour le construire, on peut utiliser la plupart des opérateurs : = <> .. ^ . [] etc. Par exemple :

```
[integer, name]
indexed=integer..integer
'sum'(indexed)
```

faire ?type[structured] pour avoir la syntaxe exacte.

On peut aussi rajouter des types :

- type procédure : on définit une procédure 'type/<type>' qui retournera true ou false. C'est encore un exemple d'interface utilisateur (voir 8). En fait :

```
type(<expr>, <type>, <>)
```

réalise l'appel :

```
'type/<type>'(<expr>, <>)
```

- type assigné : on définit

```
'type/<type>' := <type structuré>
```

Par exemple :

```
> 'type/intpol' := polynom(integer):
```

```
> 'type/logsumsin' := log('+(sin(anything)));
```

```
type/logsumsin := ln(sin(anything))
```

La fonction type permet de tester le type d'une expression :

```
type(<expr>, <type>, <arg1>, ..., <argn>)
```

retourne true ou false. <expr> est l'expression à tester et <arg1>, ..., <argn> sont des arguments optionnels selon le type.

<type> peut être un type système, un type structuré, un type procédure ou un type assigné.

**Attention**, si l'on teste le type d'un nom, celui-ci ne doit pas avoir pour valeur une suite d'expressions, car on obtient une erreur :

```
> s:=1,2,3;
```

```
s := 1, 2, 3
```

```
> type(s, list);
```

```
Error, wrong number (or type) of parameters in function type
```

La fonction whattype donne le type de données d'une expression : parmi les valeurs possibles retournées, il peut y avoir exprseq (suite d'expressions) qui est bien un type de données, mais n'est pas un type.

```
> whattype(x+y);
```

```
+
```

```
> whattype([a,b]);
```

```
list
```

```
> whattype(a/b); # attention : representation interne
```

```
*
```

```
> whattype(a,b,c);
```

```
exprseq
```

`type(<expr>, type)` permet de savoir si `<expr>` est un type.

Se souvenir de la fonction `convert` qui peut transformer le type d'une expression en un autre.

## 10 Les structures de contrôle

Une *<suite d'instructions>* est : `<> ; ... ; <> ;`

- `if <cond> then <suite d'instructions> fi`
- `if <cond> then <suite d'instructions> else <suite d'instructions> fi`

La structure suivante permet de réaliser un « case » :

```

if <cond> then <suite d'instructions>
 elif <cond> then <suite d'instructions>
 elif <cond> then <suite d'instructions>
 :
 else <suite d'instructions>
fi

```

Le dernier `else` peut être omis.

Chaque `if` doit être fermé par un `fi`.

- Si aucune expression n'est évaluée dans le `if`, le résultat est la suite d'expressions nulle : `NULL`.
- La condition `<cond>` doit pouvoir être évaluée et donner `true` ou `false` sinon une erreur apparaît :

```
> if a>b then print(a); fi;
```

Error, cannot determine if this expression is true or false: b-a < 0

Pour régler ce problème, il faut faire appel à `evalb` qui retourne `true`, `false` ou l'expression s'il ne sait pas l'évaluer :

```
> if evalb(a>b)=true then print(a); fi;
```

## 11 Les boucles

### 11.1 La boucle standard for

Plusieurs syntaxes sont possibles.

- `|for<nom>|from<expr1>|by<expr2>|to<expr3>|while<cond>|do<suite d'instructions>od;`

Toutes les expressions entre `|` peuvent être omises dans la première ligne.

Si le `from<expr1>` ou le `by<expr2>` sont omis, `<expr1>` ou `<expr2>` valent 1 par défaut.

- `|for<nom>in<expr>|while<cond>|do<suite d'instructions>od;`

`<nom>` prend successivement pour valeurs la suite des opérandes de `<expr>`.

**Attention** au piège suivant. On suppose que `l` aura comme valeur une suite d'expressions et on fait une boucle sur ses opérandes :

```
> l:=a+b,b;
> for i in l do print(i); od;
```

```

a + b
b

```

Mais `l` peut se réduire à une seule expression et on obtient un résultat indésirable ou une erreur ;

```
> l:=a+b:
> for i in l do print(i); od;
 a
 b
```

Il faut donc toujours dans ce cas utiliser `[1]` et non `l` :

```
> l:=a+b,b:
> for i in [1] do print(i); od;
 a + b
 b
```

```
> l:=a+b:
> for i in [1] do print(i); od;
 a + b
```

- `break` permet de sortir de la boucle la plus intérieure où l'on se trouve.
- `next` permet de passer à l'itération suivante dans la boucle la plus intérieure où l'on se trouve.

## 11.2 Les autres types de boucles

Les fonctions décrites ici permettent en fait de réaliser des boucles sans utiliser `for`. Elles sont à utiliser en priorité car l'utilisation de `for` est très coûteuse.

### 11.2.1 La fonction `seq`

Elle permet de construire des suites d'expressions et donc de réaliser des itérations en général beaucoup plus efficaces qu'en utilisant la boucle standard.

Les deux syntaxes sont :

```
seq(<expr>, <nom>=<expr1>..<expr2>)
seq(<expr>, <nom>=<expr1>)
```

la première construit la suite d'expressions formée de `<expr>` où `<nom>` prend respectivement toutes les valeurs de l'intervalle `<expr1>..<expr2>` et la deuxième construit la suite d'expressions formée de `<expr>` où `<nom>` prend respectivement pour valeur tous les opérandes de `<expr1>`.

Par exemple :

```
> seq(f(i), i=1..5) ;
 f(1), f(2), f(3), f(4), f(5)
> seq(i^3, i=1..10) ;
 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000
> seq(1/i, i=1+x+x^2+x^4);
 1, 1/x, 1/x^2, 1/x^4
```

Il est à noter que `seq` n'évalue pas son premier argument, à la différence de l'opérateur `$` (voir 4.6) :

```
> a||i$i=1..10;
 ai, ai, ai, ai, ai, ai, ai, ai, ai, ai
```

```
> 'a||i'$i=1..10;
 a1, a2, a3, a4, a5, a6, a7, a8, a9, a10
> seq(a||i,i=1..10);
 a1, a2, a3, a4, a5, a6, a7, a8, a9, a10
```

### 11.2.2 Les fonctions add et mul

Ces fonctions permettent respectivement de calculer des sommes et des produits, la syntaxe étant la même que pour la fonction `seq` :

```
> add(i^2,i=[a,b,c,d]);
 a2 + b2 + c2 + d2
> add(i^2,i=a*b*c);
 a2 + b2 + c2
> mul(x-i,i=1..5);
 (x - 1)(x - 2)(x - 3)(x - 4)(x - 5)
```

Comme `seq`, `add` et `mul` n'évaluent pas leur premier argument.

### 11.2.3 Les fonctions map et zip

`map` permet d'appliquer une fonction aux opérandes d'une expression :

```
> map(sin,a+b+c);
 sin(a) + sin(b) + sin(c)
> map(f,[a,b,c],2);
 [f(a, 2), f(b, 2), f(c, 2)]
```

`zip` permet d'appliquer une fonction binaire à deux listes ou deux vecteurs :

```
> zip((x,y)->x+y,[a1,a2,a3],[b1,b2,b3]);
 [a1 + b1, a2 + b2, a3 + b3]
```

## 12 Tables, tableaux, algèbre linéaire

### 12.1 Tables

Une table est un objet très général créé implicitement :

```
> t[foo,1+x^2]:=1:
> print(t);
 table([(foo, 1 + x2) = 1])
```

ou explicitement :

```
> t:=table():
> print(t);
 table([])
> t[foo,1+x^2]:=1:
> print(t);
 table([(foo, 1 + x2) = 1])
> t:=table([a,b,c]):
```

```

> print(t);
 table([1 = a, 2 = b, 3 = c])
> t:=table([(foo,bar)=1,x^2=y]):
> print(t);
 table([(foo, bar) = 1, x^2 = y])

```

La fonction `print` permet d'imprimer la table et `eval(t)` retourne l'objet table. Lors de l'impression l'ordre des éléments est imprévisible.

L'opérateur de sélection `[...]` permet de récupérer ou de modifier les éléments d'une table.

On peut avoir des tables de tables :

```

> t[x^2][x,y]:=1:
> print(t);
 table([x^2 = table([(x, y) = 1])])

```

La fonction `copy` permet de dupliquer des tables.

## 12.2 Tableaux

Un tableau est une table ayant des indices entiers prenant des valeurs dans un intervalle déterminé à l'avance et un nombre de dimensions fixé aussi à l'avance (une vérification des indices est faite lors de l'utilisation). Son type est `array`.

On utilise la fonction `array` pour créer un tableau. Plusieurs syntaxes sont utilisées. Par exemple :

```

> A:=array(0..3):
> print(A);
 array(0..3, [
 (0) = A0
 (1) = A1
 (2) = A2
 (3) = A3
])
> A:=array([a,b,c]):
> print(A);
 [a, b, c]
> A:=array([[a,b],[c,d]]):
> print(A);
 $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$

```

Maple induit la valeur des indices selon la façon dont le tableau a été créé.

Lorsque l'on imprime un tableau, `print` donne si possible une impression en 2D.

`op(2, <objet tableau>)` donne la suite des intervalles d'indexation :

```

> A:=array([[a,b],[c,d]]):

```

```
> op(2,eval(A));
```

1..2, 1..2

Dans l'exemple ci-dessus, ne pas oublier de mettre `eval(A)` (objet tableau) et non `A`.

## 12.3 Fonctions d'indexations

On peut préciser pour une table ou un tableau une fonction d'indexation. Par défaut, il n'y en a pas (sa valeur est `NULL`).

`op(1,<table ou tableau>)` donne la fonction d'indexation.

Les fonctions d'indexation connues par Maple sont `symmetric`, `antisymmetric`, `sparse`, `diagonal`, `identity`.

On indique la fonction d'indexation comme le premier argument des fonctions `table` ou `array`.

On peut créer sa propre fonction d'indexation. Si l'on donne un nom de fonction d'indexation `<nom>` non connu par le système, chaque fois que l'on veut sélectionner un élément, un appel à la fonction '`index/<nom>`' est réalisé avec comme arguments :

- le nom de l'objet (table ou tableau);
- une liste contenant l'index (`[1,2,3]` par exemple);
- une valeur logique qui est `true` ou `false` selon que l'expression est évaluée dans le membre de gauche ou de droite respectivement d'une assignation.

La valeur retournée est celle retournée par la fonction d'indexation.

## 12.4 Algèbre linéaire, manipulation de matrices

### 12.4.1 L'ancien package `linalg`

Pour manipuler matrices (type `matrix`) et vecteurs (type `vector`), il existe le package `linalg` dans lequel on trouve un grand nombre de fonctions pour les opérations sur les matrices et les vecteurs.

La fonction `vector` permet de créer des vecteurs. Ces vecteurs sont considérés dans les calculs comme des vecteurs colonnes, même si cela n'apparaît pas à l'impression. Pour transformer un vecteur en matrice, on utilise la fonction `convert`.

```
> v:=vector([x[1],x[2],x[3]]);
```

$$v := [x_1, x_2, x_3]$$

```
> convert(v,matrix);
```

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

La fonction `matrix` permet de créer facilement des matrices. Elle admet comme argument optionnel une fonction qui permet de créer les éléments de la matrice.

```
> A:=matrix(2,2,(i,j) -> x^(i+j-1));
```

$$A := \begin{bmatrix} x & x^2 \\ x^2 & x^3 \end{bmatrix}$$

Sans faire appel à ce package, on peut utiliser la fonction `evalm` qui permet d'évaluer des expressions algébriques matricielles :

`+` et `-` représentent l'addition et la soustraction matricielle,

\* représente le produit par un scalaire,  
 &\* représente la multiplication matricielle,  
 ^ représente l'élevation d'une matrice à une puissance entière, qui peut être négative (on a alors des inverses).

Cette syntaxe est valide à l'intérieur de n'importe quelle fonction du package `linalg`.

Par exemple :

```
> v:=vector([1,2]);
```

$$v := [1, 2]$$

```
> a:=matrix(2,2,(i,j)->x[i]^j);
```

$$a := \begin{bmatrix} x_1 & x_1^2 \\ x_2 & x_2^2 \end{bmatrix}$$

```
> evalm(transpose(v) &* a ^ 2 &* v);
```

$$x_1^2 + 5x_1^2x_2 + 2x_2x_1 + 2x_2^3 + 2x_1^3 + 2x_1^2x_2^2 + 4x_2^4$$

#### 12.4.2 Le nouveau package LinearAlgebra

Dans les dernières versions de Maple, on trouve un autre package pour réaliser les calculs matriciels, c'est le package `LinearAlgebra`. Il a le grand avantage d'utiliser la librairie de calcul numérique NAG pour réaliser les calculs numériques matriciels et il dispose aussi d'une syntaxe plus agréable à utiliser.

Par exemple, si l'on reprend les exemples du paragraphe précédent, on écrira :

```
> v:=<x[1] | x[2] | x[3]>;
```

$$v := [x_1, x_2, x_3]$$

```
> v:=<x[1], x[2], x[3]>;
```

$$v := \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

```
> m:=<<1,2> | <3,4>>;
```

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

pour avoir respectivement un vecteur ligne, un vecteur colonne ou une matrice.

On pourra aussi utiliser la fonction `Matrix` :

```
> A:=Matrix(2,2,(i,j) -> x^(i+j-1));
```

$$A := \begin{bmatrix} x & x^2 \\ x^2 & x^3 \end{bmatrix}$$

Mais maintenant on peut utiliser les opérateurs d'addition `+`, de multiplication matricielle `.` et de multiplication par un scalaire `*` directement sur ces vecteurs et ces matrices :

```
> with(LinearAlgebra):
```

```
> v:=<1,2>;
```

$$v := \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

```
> a:=Matrix(2,2,(i,j)->x[i]^j);
```

$$a := \begin{bmatrix} x_1 & x_1^2 \\ x_2 & x_2^2 \end{bmatrix}$$

```
> Transpose(v) . a ^ 2 . v;
```

$$x_1^2 + 5x_1^2x_2 + 2x_2x_1 + 2x_2^3 + 2x_1^3 + 2x_1^2x_2^2 + 4x_2^4$$

## 13 Procédures

Maple utilise le nom de « procédure » pour dénommer ce qui correspond généralement à des « fonctions » dans la terminologie d'un langage comme PASCAL par exemple. Nous emploierons les deux termes indifféremment.

### 13.1 Opérateur flèche

Le moyen le plus simple de définir une procédure qui calcule une expression algébrique est d'utiliser la notation flèche. Par exemple, la fonction qui à  $(x, y)$  associe  $x^2 + y^2$  peut être définie par l'expression Maple  $(x, y) \rightarrow x^2 + y^2$ . On peut donner un nom à cette expression et l'utiliser alors de la façon habituelle :

```
> ((x, y) -> x^2 + y^2)(2, 3);
```

13

```
> f := (x, y) -> x^2 + y^2;
```

$$f := (x, y) \rightarrow x^2 + y^2$$

```
> f(2, 3);
```

13

Réciproquement, il est possible d'obtenir une fonction à partir d'une expression algébrique en utilisant la fonction Maple `unapply` :

```
> e := 1 - sin(x)^2 + cos(x)^3;
```

$$e := 1 - \sin(x)^2 + \cos(x)^3$$

```
> f := unapply(e, x);
```

$$f := x \rightarrow 1 - \sin(x)^2 + \cos(x)^3$$

```
> f(0);
```

2

**Attention**, lorsque l'on veut transformer une expression en procédure de type flèche, il est impératif d'utiliser la fonction `unapply`. En effet, ce qui suit ne marche pas car ce qui se trouve après la flèche n'est pas évalué lors de la définition de la procédure :

```
> e := x + sin(x);
```

$$e := x + \sin(x)$$

```
> f := x -> e;
```

$$f := x \rightarrow e$$

```
> f(1);
```

$$x + \sin(x)$$

Enfin, l'opérateur `@` permet de composer de telles fonctions, l'opérateur `@@` est la composition itérée et l'opérateur `D` permet de différencier :

```

> f:=x->1+1/x;
 f := x → 1 + 1/x
> g:=x->1+x^2;
 g := x → 1 + x^2
> (f@g)(x);
 1 + 1/(1 + x^2)
> D(f);
 x → -1/x^2

```

### 13.2 Définition d'une procédure

La définition d'une procédure est aussi une expression.

Syntaxe (simplifiée) définissant l'objet procédure :

```

proc(<suite de noms avec types>)
 global <suite de noms>;
 local <suite de noms>;
 options <suite de noms>;
 <suite d'instructions>
end

```

*<suite de noms avec types>* est de la forme *<arg1> : :<type1>, ..., <argn> : :<typen>* où *<argi>* est le ième argument et *: :<typei>* est optionnel et décrit le type de l'argument, cela permettant de réaliser une vérification à l'appel de la procédure. Il est fortement recommandé d'utiliser ces arguments optionnels chaque fois que cela est possible.

Les trois déclarations **global**, **local** et **options** sont optionnelles.

La valeur retournée par la procédure est celle de la dernière instruction évaluée (sauf utilisation de fonctions spéciales que l'on verra plus loin).

```

> (proc(x) 1+x; end) (2);
 3

```

On peut assigner l'objet procédure à un nom. C'est la façon habituelle de définir une procédure :

```

> f:= proc(x) 1+x; end;
> f(2);
 3

```

Ne pas confondre le nom de la procédure **f** et l'objet procédure **eval(f)** (voir 6.3).

**print(f)** ; permet d'imprimer la procédure.

### 13.3 Arguments de la procédure

On peut avoir plus d'arguments dans l'appel de la procédure que le nombre d'arguments lors de la définition de la procédure. Les noms spéciaux suivants sont définis à l'intérieur d'une procédure :

**args** est la suite d'expressions des arguments : **args[<i>]** est donc l'argument numéro *<i>*.  
**nargs** est le nombre d'arguments lors de l'appel de la procédure.

Cela permet de faire référence à des arguments supplémentaires et donc de traiter le cas des procédures avec un nombre d'arguments variable.

`op(1,eval(<f>))` retourne la suite des arguments lors de la définition de la procédure `<f>` :

```
> f:=proc(a,b) a+b end;
> op(1,eval(f));
```

$a, b$

### 13.4 Mécanisme d'évaluation des arguments

Après que les arguments ont été évalués lors de l'appel de la procédure, toute occurrence d'un argument dans le corps de la procédure est remplacé par la valeur trouvée **avant** l'exécution de la procédure (« call by evaluated name ») et **n'est plus évalué**.

Cela implique que de façon générale, on ne peut pas assigner de valeur à un argument à l'intérieur du corps de la procédure, sauf dans le cas suivant :

on peut faire retourner à une procédure plus d'une valeur par l'intermédiaire des arguments à condition de les quoter (cela fonctionne aussi si l'argument n'a pas de valeur).

Exemple d'une procédure retournant la liste des opérandes d'une expression et le nombre d'opérandes :

```
> f:=proc(e,n) n:=nops(e); op(e); end;
 f := proc(e, n) n := nops(e); op(e) end proc
```

```
> f(a+b,p);
```

$a, b$

```
> p;
```

2

```
> f(a+b,'p');
```

$a, b$

```
> p;
```

2

```
> f(a+b,p);
```

```
Error, (in f) illegal use of a formal parameter
```

Remarquer que lors du premier appel on n'a pas coté `p` et il n'y a pas eu de problème car il n'avait pas de valeur. On voit plus loin que lorsqu'il a une valeur, il y a une erreur.

Ce mécanisme d'évaluation peut entraîner l'erreur suivante, dans l'exemple d'une procédure retournant la somme des opérandes d'une expression et le nombre d'opérandes :

```
> f:=proc(e,n)
> local i;
> n:=nops(e);
> sum(op(i,e),i=1..n);
> end;
 f := proc(e, n) local i; n := nops(e); sum(op(i, e), i = 1..n) end proc
> f(a*b,'p'),p;
```

$$\sum_{i=1}^p \text{op}(i, a b), 2$$

`f(a*b, 'p')` ; ne fonctionne pas car dans `1..n`, `n` vaut `p` et non `nops(a+b)`.

Il faut donc utiliser une variable locale supplémentaire :

```
> f:=proc(e,n)
> local i,nn;
> nn:=nops(e);
> n:=nn;
> sum(op(i,e),i=1..nn);
> end;
f := proc(e, n) local i, nn; nn := nops(e); n := nn; sum(op(i, e), i = 1..nn) end proc
> f(a*b, 'p'),p;
```

$a + b, 2$

ou bien utiliser la fonction `eval` :

```
> f:=proc(e,n)
> local i;
> n:=nops(e);
> sum(op(i,e),i=1..eval(n));
> end;
f := proc(e, n) local i; n := nops(e); sum(op(i, e), i = 1..eval(n)) end proc
> f(a*b, 'p'),p;
```

$a + b, 2$

### 13.5 Variables locales et variables globales

Les variables locales à la procédure sont définies par le `local` optionnel au début de la définition de la procédure.

Les variables globales dans la procédure sont définies par le `global` optionnel au début de la définition de la procédure.

Il est fortement recommandé de déclarer ainsi explicitement toutes les variables qui apparaissent dans le corps de la procédure.

Toute variable qui n'est pas déclarée est considérée comme locale ou globale selon la façon dont elle est employée dans la procédure ; faire `?procedures` pour avoir la règle.

`op(2,eval(<f>))` retourne la suite des variables locales de la procédure `<f>` :

```
> f:=proc(x) local i,j; i:=x^2; j:=x^3; i+j; end;
f := proc(x) local i, j; i := x^2; j := x^3; i + j end proc
> op(2,eval(f));
```

$i, j$

Lors de l'exécution de la procédure, les variables locales ne sont évaluées qu'à un seul niveau (« one-level evaluation mode »). Si l'on veut le mode d'évaluation complet (« full evaluation »), on utilise la fonction `eval`.

### 13.6 Procédures imbriquées

Il est maintenant possible (depuis cette release 5 de Maple) de définir des procédures à l'intérieur de procédures. Dans les releases antérieures, une procédure définie à l'intérieur d'une autre procédure était considérée comme définie au toplevel.

Il y a maintenant la notion de fermeture lexicale (lexical closure) qui permet entre autres d'utiliser les variables locales de la procédure dans laquelle notre procédure est définie. Pour avoir toutes les subtilités et la complexité de ces notions, faire `?examples,lexical`.

### 13.7 Table de remember

Un certain nombre d'options peuvent être indiquées à l'aide du option optionnel au début de la définition de la procédure.

`op(3,eval(<f>))` retourne la suite des options.

On va décrire l'option `remember`.

À chaque fonction peut être associée une « table de remember » qui est une table gardant les résultats des calculs d'une fonction.

Certaines fonctions Maple comme `evalf`, `taylor`, `divide`, `normal`, `expand`, `diff`, `readlib` et `frontend` le font par défaut. Comparer par exemple les temps de calcul de deux appels successifs à `diff` pour une expression compliquée.

`op(4,eval(<f>))` retourne la table de remember associée à la procédure. Ce peut être NULL s'il n'y en a pas.

Si l'utilisateur utilise l'option `remember`, tous les calculs effectués par la procédure seront gardés dans la table de remember associée ainsi créée :

```
> fac:=proc(n::nonnegint)
> option remember;
> if n=0 then 1 else n*fac(n-1) fi
> end;
```

```
fac := proc(n :: nonnegint)
option remember;
if n = 0 then 1 else n * fac(n - 1) end if
end proc
```

```
> t:=time(): fac(1000): time()-t;
0.030
> t:=time(): fac(1100): time()-t;
0.011
```

Dans l'exemple précédent, le temps de calcul de `fac(1100)` est plus court que celui de `fac(1000)`.

On peut aussi utiliser l'opérateur d'assignation pour mettre des valeurs dans la table de remember (ceci indépendamment de l'option `remember`). Par exemple :

```
fac:=proc(n::nonnegint)
option remember;
n*fac(n-1);
end;
fac(0):=1;
```

Il faut faire l'assignation après la définition de la fonction.

Cette assignation permet de définir facilement les valeurs initiales pour des suites, et aussi de faire quelques simplifications de base pour des procédures, par exemple  $f(0) := 0$  ;

Pour supprimer des valeurs à l'intérieur de la table de remember, on peut le faire par accès direct à la table :

```
> f:=proc(x) option remember; 1+x; end:
> f(1):
> t:=op(4,eval(f)):
> print(t);
 table([1 = 2])
> t[1] := 't[1]':
> t:=op(4,eval(f)):
> print(t);
 table([])
```

On peut aussi utiliser la fonction `forget`.

### 13.8 Retour explicite d'une procédure

`RETURN(<suite d'expressions>)` permet de sortir de la procédure et la valeur retournée est celle de la suite d'expressions.

`ERROR(<suite d'expressions>)` permet de sortir de la procédure et de retourner au toplevel avec un message d'erreur.

### 13.9 Récupération d'erreur

`traperror(<expression>)` évalue `<expression>` et retourne sa valeur s'il n'y a pas d'erreur, sinon retourne le message d'erreur, c'est-à-dire la chaîne de caractères après `ERROR` (erreur venant d'une fonction système ou retournée par la fonction `ERROR`).

`lasterror` est une variable globale qui donne la valeur du dernier message d'erreur.

Tout cela permet de récupérer une erreur et éventuellement de continuer selon qu'il y a ou non erreur :

```
exprval :=traperror(<expression>);
if exprval <> lasterror
 then<> ← il n'y a pas eu erreur et la valeur
 de l'expression est dans exprval
 else<> ← il y a eu erreur et le message
 d'erreur est dans lasterror
fi;
```

## 14 Fichiers, entrées/sorties, sauvegardes

On peut faire des impressions sur un fichier au lieu du terminal. Après la commande `writeto(<fichier>)` tout ce qui suit est écrit dans le fichier `<fichier>` jusqu'au prochain `writeto`. `writeto(terminal)` permet à nouveau d'écrire sur le terminal.

Une autre méthode est d'utiliser les fonctions `writeline` et `fprintf`.

Pour lire des fichiers de données, on peut utiliser les fonctions `readline` et `fscanf`.

On peut lire des fichiers pour les charger dans Maple comme des lignes de commandes.

Selon l'extension des noms de fichiers, les fichiers sont considérés comme étant en mode caractères (par exemple en ASCII sous UNIX) ou en format interne Maple. Dans ce dernier cas, leur nom doit se terminer par `.m`, par exemple `toto.m`.

Le mot-clé `read` permet de lire ces fichiers et de les charger dans Maple. Il reconnaît par l'extension si c'est un fichier en mode caractères ou en format interne.

On peut sauver des variables ou l'état complet d'une session avec le mot-clé `save` :

`save <fichier>` sauve toutes les variables de la session dans le fichier `<fichier>`

`save <suite d'expressions>, <fichier>` sauve les valeurs de la suite d'expressions dans le fichier `<fichier>`.

Ici aussi, selon l'extension de `<fichier>`, ce sera sauvé en mode caractères ou en format interne.

## 15 Debugging

Il existe un débogueur intégré à Maple. On peut l'appeler de différentes façons, par exemple en utilisant les fonctions `stopat` ou `DEBUG`. On a alors les fonctionnalités classiques des outils de ce type : exécution pas à pas, arrêt sous condition lors de l'exécution des procédures, etc.

Faire `?debugger` pour avoir la description du fonctionnement du débogueur.

On peut aussi tracer des fonctions avec la commande `trace` et supprimer ce traçage avec la commande `untrace`.

**Attention** la fonction `trace` existe aussi dans le package `linalg`, donc si l'on fait `with(linalg)` la fonction de debugging `trace` est effacée. Il faut la sauvegarder avant, par exemple :

```
> Trace:=op(trace):
```

On peut de toute façon retrouver la fonction `trace` effacée en faisant `readlib(trace);`.

## 16 Librairies, packages

### 16.1 Les différents types de fonctions

En fait on peut classer les fonctions Maple en 3 catégories :

1. Les « builtin » fonctions, internes au système Maple et qui se trouvent dans le noyau pour des raisons d'efficacité ; on n'a pas accès à leur source. C'est le cas par exemple de la fonction `normal`.
2. Les fonctions disponibles directement à l'utilisateur. Ce sont les plus nombreuses. C'est le cas par exemple de la fonction `int`. En fait, elles sont chargées automatiquement lors de leur premier appel par la fonction `readlib`.

L'exécution de `readlib(<nom>)` a pour effet de :

- lire le fichier  $\langle nom \rangle.m$  dans la librairie Maple (valeur de `libname`). Dans cette lecture une valeur doit être assigné à  $\langle nom \rangle$  (sinon c'est une erreur).
- retourner la valeur assignée à  $\langle nom \rangle$ .

Ces fonctions disponibles directement sont définies par : `f := 'readlib('f')'` ;

3. Les fonctions qui sont définies dans des packages.

On a accès au source de toutes les fonctions, sauf celles du type « builtin ». En effet, on peut toujours imprimer le source de ces fonctions en utilisant la fonction `print` après avoir fait `interface(verboseproc=2)`. Cela représente la plupart des fonctions. Noter que l'on peut aussi utiliser aussi directement la fonction `showstat` (les lignes sont alors numérotées).

## 16.2 Les packages

Il en existe 71 en Maple 8.

En fait le fonctionnement d'un package est le suivant : si `p` est le nom d'un package contenant les fonctions `f1` et `f2`, une table `p` est définie avec :

```
p[f1] := 'readlib('p/f1')';
p[f2] := 'readlib('p/f2')';
```

(ou bien `:=` l'expression de la fonction si l'on crée son propre package).

La commande `with(p)` réalise simplement les assignations :

```
f1:=p[f1];
f2:=p[f2];
```

Il est donc facile de créer son propre package.

Pour utiliser le package `linalg` par exemple :

- `with(linalg)` charge tout le package et on peut utiliser les noms des fonctions directement, par exemple `transpose(a)`
- on peut utiliser la table et ne pas charger tout le package, par exemple :  
`linalg[transpose](a)`

## 17 Graphique

On peut utiliser Maple pour tracer des courbes et des surfaces. Un grand nombre de fonctions utilisées pour les tracés se trouvent dans le package `plots` que l'on va supposer chargé une fois pour toutes par la commande `with(plots)` :

Pour tracer des courbes en 2 dimensions, la fonction de base est `plot`.

On peut tracer :

- des courbes définies par  $y = f(x)$  en coordonnées cartésiennes ou par  $\rho = f(\theta)$  en coordonnées polaires :  
`plot(sin(x), x=0..Pi);`  
`polarplot(cos(theta/3), theta=0..6*Pi);`
- des courbes définies paramétriquement par  $(x = f(t) \quad y = g(t))$  en coordonnées cartésiennes ou par  $(\rho = f(t) \quad \theta = g(t))$  en coordonnées polaires :  
`plot([2*cos(t)-cos(2*t), 2*sin(t)-sin(2*t)], t=0..2*Pi);`  
`polarplot([cos(theta), sin(theta)], theta=0..4*Pi);`
- des listes de points  $(x_1, y_1)(x_2, y_2) \cdots (x_n, y_n)$  :  
`plot([[3,0], [0,1], [1,1], [2,2], [1,2], [2,3], [3,2]], style=LINE);`

- des fonctions définies implicitement par  $f(x, y) = 0$  :  
`implicitplot(x^2+y^2=1,x=-1..1,y=-1..1);`

En 3 dimensions, on peut tracer :

- des surfaces définies par  $z = f(x, y)$  en coordonnées cartésiennes :  
`plot3d(sin(x+y),x=-1..1,y=-1..1);`
- des surfaces définies paramétriquement par  $(x = f(s, t) \quad y = g(s, t) \quad z = h(s, t))$  :  
`plot3d([s*sin(s)*cos(t),s*cos(s)*cos(t),s*sin(t)],  
s=0..2*Pi,t=0..Pi);`
- des surfaces définies implicitement par  $f(x, y, z) = 0$  :  
`implicitplot3d(x^3+y^3+z^3=-1,x=-2..2,y=-2..2,z=-2..2);`
- des courbes gauches définies paramétriquement par  $(x = f(t) \quad y = g(t) \quad z = h(t))$  :  
`spacecurve([cos(t),sin(t),t],t=0..4*Pi);`

On peut aussi faire des tracés en coordonnées cylindriques et sphériques.

Ces fonctions de tracé prennent un très grand nombre d'arguments optionnels qui permettent de modifier le style du tracé, les axes de coordonnées, l'échelle et le point de vue en 3 dimensions.

Il est possible d'assigner à des noms des tracés de courbes, sans les afficher, et ensuite seulement de les afficher en utilisant la fonction `display`. Cette dernière permet aussi de superposer plusieurs courbes précédemment sauveées dans des variables.

On peut aussi tracer des histogrammes, d'avoir en 3 dimensions des tracés avec surfaces cachées ou non, avec des équipotentielles, des nuages de points...

Il est possible de réaliser des animations à l'aides des fonctions `animate`, `animate3d` et `display`.

Enfin, le package `plottools` définit un certain nombre d'objets graphiques, par exemple des ellipses, des sphères, des octaèdres, ... que l'on peut ensuite tracer à l'aide de la fonction `display`.

## Quelques livres

Les livres de référence sur Maple sont les deux ouvrages des développeurs publiés chez Springer-Verlag : *Maple V, Learning Guide* et *Maple V, Programming Guide*.

Une bonne introduction aux mathématiques du calcul formel :  
DAVENPORT (James H.), SIRET (Yves), et TOURNIER (Évelyne). – *Calcul formel*. – Masson, Paris, 1993, seconde édition.

Un très bon livre qui décrit les algorithmes du calcul formel :  
VON ZUR GATHEN (Joachim) and GERHARD (Jürgen) . – *Modern Computer Algebra*. – Cambridge University Press, 1999.

Notre livre décrit de façon détaillée l'utilisation du calcul formel dans la plupart des domaines des mathématiques :

GOMEZ (Claude), SALVY (Bruno), et ZIMMERMANN (Paul). – *Calcul formel : mode d'emploi, exemples en Maple*. (2<sup>e</sup> tirage avec mise à jour Maple V.4) – Masson, Paris, 1995.

La plupart des exercices du livre ont été corrigés par P. Dumas et se trouvent sur le site Web :  
<http://algo.inria.fr/dumas/GoSaZi95/Answers>

On peut trouver une liste des livres écrits sur Maple à partir du site Web de WMI à l'adresse :  
<http://www.maplesoft.com/publications/books>

## Sites WEB sur le calcul formel

Site de Waterloo Maple Inc. : <http://www.maplesoft.com>

Site du centre de calcul formel Médecis : <http://www.medicis.polytechnique.fr>

Site du CAIN (Computer Algebra Information Network) Europe :  
<http://www.can.nl/cain.html>

## Table des matières

|           |                                                                 |           |
|-----------|-----------------------------------------------------------------|-----------|
| <b>1</b>  | <b>Introduction</b>                                             | <b>1</b>  |
| <b>2</b>  | <b>Déroulement d'une session</b>                                | <b>1</b>  |
| <b>3</b>  | <b>Commandes utiles</b>                                         | <b>1</b>  |
| <b>4</b>  | <b>Expressions</b>                                              | <b>2</b>  |
| 4.1       | Constantes . . . . .                                            | 2         |
| 4.2       | Chaînes de caractères . . . . .                                 | 2         |
| 4.3       | Noms . . . . .                                                  | 2         |
| 4.4       | Opérateurs . . . . .                                            | 3         |
| 4.5       | Intervalle (range) . . . . .                                    | 3         |
| 4.6       | Suite d'expressions (sequence ou expression sequence) . . . . . | 3         |
| 4.7       | Ensembles, listes . . . . .                                     | 4         |
| <b>5</b>  | <b>Les fonctions op, nops et subsop</b>                         | <b>6</b>  |
| <b>6</b>  | <b>L'évaluation dans Maple</b>                                  | <b>7</b>  |
| 6.1       | Les noms . . . . .                                              | 7         |
| 6.2       | Les opérateurs . . . . .                                        | 9         |
| 6.3       | Les expressions fonctionnelles . . . . .                        | 9         |
| 6.4       | Les tables et les tableaux . . . . .                            | 11        |
| 6.5       | Les évaluateurs de Maple . . . . .                              | 12        |
| <b>7</b>  | <b>La simplification dans Maple</b>                             | <b>12</b> |
| <b>8</b>  | <b>Modifier le comportement de Maple</b>                        | <b>14</b> |
| <b>9</b>  | <b>Les types</b>                                                | <b>14</b> |
| <b>10</b> | <b>Les structures de contrôle</b>                               | <b>17</b> |
| <b>11</b> | <b>Les boucles</b>                                              | <b>17</b> |
| 11.1      | La boucle standard <code>for</code> . . . . .                   | 17        |
| 11.2      | Les autres types de boucles . . . . .                           | 18        |
| 11.2.1    | La fonction <code>seq</code> . . . . .                          | 18        |
| 11.2.2    | Les fonctions <code>add</code> et <code>mul</code> . . . . .    | 19        |
| 11.2.3    | Les fonctions <code>map</code> et <code>zip</code> . . . . .    | 19        |
| <b>12</b> | <b>Tables, tableaux, algèbre linéaire</b>                       | <b>19</b> |
| 12.1      | Tables . . . . .                                                | 19        |
| 12.2      | Tableaux . . . . .                                              | 20        |
| 12.3      | Fonctions d'indexations . . . . .                               | 21        |
| 12.4      | Algèbre linéaire, manipulation de matrices . . . . .            | 21        |
| 12.4.1    | L'ancien package <code>linalg</code> . . . . .                  | 21        |
| 12.4.2    | Le nouveau package <code>LinearAlgebra</code> . . . . .         | 22        |

|                                                        |           |
|--------------------------------------------------------|-----------|
| <i>Notes de cours Maple</i>                            | 34        |
| <b>13 Procédures</b>                                   | <b>23</b> |
| 13.1 Opérateur flèche . . . . .                        | 23        |
| 13.2 Définition d'une procédure . . . . .              | 24        |
| 13.3 Arguments de la procédure . . . . .               | 24        |
| 13.4 Mécanisme d'évaluation des arguments . . . . .    | 25        |
| 13.5 Variables locales et variables globales . . . . . | 26        |
| 13.6 Procédures imbriquées . . . . .                   | 27        |
| 13.7 Table de remember . . . . .                       | 27        |
| 13.8 Retour explicite d'une procédure . . . . .        | 28        |
| 13.9 Récupération d'erreur . . . . .                   | 28        |
| <b>14 Fichiers, entrées/sorties, sauvegardes</b>       | <b>29</b> |
| <b>15 Debugging</b>                                    | <b>29</b> |
| <b>16 Librairies, packages</b>                         | <b>29</b> |
| 16.1 Les différents types de fonctions . . . . .       | 29        |
| 16.2 Les packages . . . . .                            | 30        |
| <b>17 Graphique</b>                                    | <b>30</b> |
| <b>Bibliographie</b>                                   | <b>32</b> |