

2. Les principaux éléments du langage

2.1. NOMS ET VARIABLES

- 2.1.1. Les noms dans Maple
- 2.1.2. Utilisation des caractères ‘
- 2.1.3. Passage à la ligne dans un nom
- 2.1.4. Noms de variable
- 2.1.5. Assignation et désassignation
- 2.1.6. Noms obtenus par concaténation
- 2.1.7. Lettres grecques
- 2.1.8. Évaluation récursive d'une expression
- 2.1.9. Test 1
- 2.1.10. Réponses au test 1

2.2. NOMBRES ET CONSTANTES

- 2.2.1. Nombres entiers
- 2.2.2. Nombres rationnels
- 2.2.3. Nombres réels
- 2.2.4. Évaluation forcée au format réel
- 2.2.5. Nombres complexes
- 2.2.6. Constantes symboliques
- 2.2.7. Expressions constantes
- 2.2.8. Fonctions avec arguments constants
- 2.2.9. Test 2
- 2.2.10. Réponses au test 2

2.3. OPÉRATEURS

- 2.3.1. Types d'opérateurs
- 2.3.2. Opérateurs arithmétiques
- 2.3.3. Opérateurs logiques
- 2.3.4. Opérateurs de relation
- 2.3.5. Autres types d'opérateurs
- 2.3.6. Questions de priorité
- 2.3.7. Test 3
- 2.3.8. Réponses au test 3

2.4. FONCTIONS

- 2.4.1. Les fonctions dans Maple
- 2.4.2. Opérations sur les fonctions
- 2.4.3. Définir des fonctions
- 2.4.4. L'instruction unapply
- 2.4.5. Test 4
- 2.4.6. Réponses au test 4

2.5. EXPRESSIONS

- 2.5.1. Arborescence d'une expression
- 2.5.2. Les fonctions op et nops
- 2.5.3. Remplacements dans une expression
- 2.5.4. Recherches dans une expression
- 2.5.5. Mapper une fonction dans une expression
- 2.5.6. Reconnaissance de motif

2.6. EXPRESSIONS : EXEMPLES

- 2.6.1. Les fonctions op et nops
- 2.6.2. Remplacements par subs
- 2.6.3. Remplacements par subsop
- 2.6.4. Recherches dans une expression
- 2.6.5. Sélections dans une expression
- 2.6.6. Mapper une fonction dans une expression
- 2.6.7. Reconnaissance de motifs

2.7. SÉQUENCES, LISTES, ENSEMBLES

- 2.7.1. Séquences
- 2.7.2. Listes
- 2.7.3. Ensembles
- 2.7.4. Passer d'une structure à l'autre
- 2.7.5. Rechercher des éléments
- 2.7.6. Ajouter, remplacer ou enlever des éléments
- 2.7.7. Mapper des opérations sur des ensembles ou des listes

2.8. SÉQUENCES, LISTES, ENSEMBLES : EXEMPLES

- 2.8.1. Séquences
- 2.8.2. Listes
- 2.8.3. Ensembles
- 2.8.4. Passer d'une structure à l'autre
- 2.8.5. Rechercher des éléments
- 2.8.6. Ajouter, remplacer ou enlever des éléments
- 2.8.7. Mapper des opérations sur des ensembles ou des listes

2.9. TABLES

- 2.9.1. Noms indicés
- 2.9.2. Création d'une table
- 2.9.3. Ajouter ou supprimer des éléments à une table
- 2.9.4. Lecture et affichage d'une table
- 2.9.5. Dupliquer une table
- 2.9.6. Fonctions d'indexation
- 2.9.7. Opérandes d'une table

2.10. TABLES : EXEMPLES

- 2.10.1. Noms indicés
- 2.10.2. Création d'une table
- 2.10.3. Évaluation vers le dernier nom
- 2.10.4. Fonctions d'indexation
- 2.10.5. Opérandes d'une table

2.11. TABLEAUX

- 2.11.1. Création d'un tableau
- 2.11.2. Lecture et affichage d'un tableau
- 2.11.3. Opérandes d'un tableau
- 2.11.4. Conversions entre listes, tables et tableaux
- 2.11.5. Opérations sur vecteurs et matrices

2.12. TABLEAUX : EXEMPLES

- 2.12.1. Création, affichage, lecture et opérandes d'un tableau
- 2.12.2. Conversion entre listes, tables et tableaux
- 2.12.3. Opérations sur vecteurs et matrices

2.13. LES TYPES D'OBJETS

2.13.1. Types de base

2.13.2. Réunions ou synonymes de types de base

2.13.3. Types "précisés"

2.13.4. Types rékursifs

2.13.5. Types structurés

2.13.6. La fonction hastype

2.14. LES TYPES D'OBJETS : EXEMPLES

2.14.1. Types de base

2.14.2. Réunions ou synonymes de types de base

2.14.3. Types précisés

2.14.4. Types rékursifs

2.14.5. Types structurés

2.14.6. La fonction hastype

2.14.7. Fonction match ou fonction type ?

2.1. NOMS, VARIABLES

2.1.1. Les noms dans Maple

Les *noms*, au sens où nous l'entendrons ici, servent à former des mots, des messages, ou à nommer des objets. On rappelle que Maple différencie les majuscules des minuscules.

Un nom débute en général par une lettre suivie éventuellement par des lettres ou des chiffres.

Le caractère de soulignement `_` peut également figurer dans un nom, à n'importe quelle position.

Mais on évitera de le placer au début car Maple utilise de tels noms pour son usage interne.

2.1.2. Utilisation des caractères ‘

On peut en fait former un nom avec des caractères quelconques notamment des espaces, ou des caractères particuliers comme `*`, `?`, `+`, etc.

Il suffit pour celà de débiter et de terminer le nom par le caractère ‘ (accent grave).

Il est même possible d'utiliser ce caractère ‘ dans le nom, à condition de le doubler en “.

Les accents graves ‘ n'apparaissent que lors de la création du nom. Ils ne sont ensuite plus affichés, et ils ne comptent pas dans la longueur du nom. Ils sont indispensables pour créer des noms contenant des caractères ni alphabétiques ni numériques (par exemple des caractères accentués!), ou débutant par un chiffre.

Dans le cas où les accents graves ne sont pas à priori indispensables, ils permettent de s'assurer que le nom sera bien interprété comme tel par Maple. Par exemple ‘`expand`’ est un nom qui ne risque pas d'être interprété comme un appel à la fonction intégrée `expand`.

Pour obtenir un accent grave au clavier, on pourra utiliser la combinaison de touches `AltGr+7` suivie d'un appui sur la barre d'espacement. Cet appui sur espace n'est vraiment indispensable que lorsque l'accent grave précède une lettre comme *a*, *e*, *i*, *o*, *u*, pour éviter que ne soit créée une lettre accentuée.

Dans tous les cas la taille maximum d'un nom est de 499 caractères.

2.1.3. Passage à la ligne dans un nom

Même si ce n'est pas très courant on peut placer un saut de ligne (par `Entrée` ou `Shift+Entrée`) dans un nom nécessairement débutant par ‘.

On n'omettra pas cependant de clore ce nom par un autre accent grave, sans quoi toute nouvelle ligne serait encore considérée comme une continuation du nom (un message de Maple permet de s'en rendre compte).

Si on arrive en fin de ligne à l'écran et si le nom n'est pas encore terminé, il y a saut à la ligne automatique, qui ne sera pas considéré comme faisant partie du nom.

On peut également forcer le passage à la ligne en utilisant le caractère `\` (qui sera ignoré dans la formation du nom). Enfin pour qu'un caractère `\` apparaisse effectivement dans un nom, il faut le doubler en `\\`.

2.1.4. Noms de variable

Il est très souvent utile de mémoriser un résultat en lui donnant un nom.

N'importe quelle chaîne de caractères peut alors être utilisée pour ce nom, mais il est recommandé, pour des raisons de lisibilité, de se limiter à des noms débutant par un caractère alphabétique (lettre) suivi éventuellement par des lettres, des chiffres ou des caractères de soulignement.

On ne peut donner un nom à un résultat si ce nom est également celui d'une instruction intégrée de Maple.

2.1.5. Assignment et désassignment

L'opération qui consiste à associer un résultat à un nom est l'*assignment* (ou encore *affectation*), qui utilise le double caractère `:=`.

La syntaxe est donc `name := expr`.

Ici *expr* peut être une expression quelconque, qui est d'abord *évaluée* avant de faire l'objet de cette assignment. Dans toute la suite de la session (sauf évidemment si on affecte un nouveau résultat à *name*), le nom *name* sera toujours remplacé par cette valeur de *expr*.

Quand une instruction `name := expr;` (bien noter le point-virgule) est exécutée, Maple affiche quelque chose comme `name := valeur_de_expr`.

Que l'instruction se termine par `;` (point-virgule) ou par `:` (deux-points), le contenu de la fonction `%` est *valeur_de_expr*.

Par exemple:

```
> x:=2^5; %+1;
```

$$x := 32$$

33

Tant qu'un nom comme *name* n'a pas subi une affectation particulière, il continue à ne désigner que lui-même. On peut alors considérer *name* comme une variable formelle (symbolique), c'est-à-dire une variable au sens mathématique habituel.

Une des grandes forces de Maple est de pouvoir traiter, manipuler et simplifier des expressions contenant de tels noms formels.

Pour désassigner un nom comme *name*, on exécutera l'instruction `name := 'name'` (on notera l'utilisation d'*apostrophes*), ce qui aura pour conséquence que le nom *name* ne pointera (jusqu'à nouvel ordre) que sur lui-même : il aura ainsi retrouvé son statut de variable formelle.

On rappelle que l'instruction `restart` permet de redémarrer complètement la session de travail et en particulier de désassigner tous les noms de variables qu'aurait créés l'utilisateur.

2.1.6. Noms obtenus par concaténation

On peut former des noms à partir d'autres noms, par *concaténation*.
L'opérateur utilisé est le point (.).

Par exemple, l'expression `x.1` sera évaluée en `x1`, et `A.i.j` sera évaluée en le nom `Aij`. Dans cette dernière construction on suppose que les variables `i` et `j` ne sont pas assignées, car sinon les noms `i` et `j` seraient d'abord évalués avant la concaténation.

Dans la création d'un nom par concaténation, le nom le plus à gauche n'est jamais évalué.

Exemple :

```
> restart:  A:=1:  i:=2:  j:=3:  A.i.j:=10; A.k:=100;
```

$$A23 := 10$$

$$Ak := 100$$

2.1.7. Lettres grecques

Les lettres grecques sont saisies sous la forme *alpha*, *beta*, *gamma*, *lambda*, *mu*, *epsilon*, *pi*, etc., mais Maple les affiche avec le caractère correspondant de l'alphabet grec.

Par exemple :

```
> alpha,beta,delta,gamma,lambda,epsilon,mu,nu,pi,phi,rho,sigma,theta;
```

$$\alpha, \beta, \delta, \gamma, \lambda, \varepsilon, \mu, \nu, \pi, \phi, \rho, \sigma, \theta$$

2.1.8. Evaluation récursive d'une expression

Quand une expression `expr` est évaluée (par exemple lors d'une assignation `name := expr`), les noms assignés qu'elle contient sont évalués. Il se peut que le contenu d'un de ces noms soit lui-même une expression contenant des noms assignés. Ceux-ci sont à leur tour évalués.

On voit donc se mettre en place une évaluation récursive, jusqu'à ce que les noms qui apparaissent ne soient plus assignés (c'est-à-dire ne désignent qu'eux-mêmes).

Par exemple :

```
> restart:  y:=3*x+1:  x:=2*z:  x+y+z;
```

$$9z + 1$$

Il y a une exception à cette règle de l'évaluation récursive complète, pour ce qui concerne les objets réputés volumineux que sont les tables et les procédures, que l'on verra plus tard. Dans ce cas l'évaluation en chaîne cesse sur le dernier nom avant que n'apparaisse la table ou la procédure.

2.1.9. Test 1

Imaginez les réponses (et précisez quand celles-ci sont des messages d'erreur) à chacune des lignes de saisie suivantes. Comparer ensuite avec les réponses effectivement données par Maple, et justifiez les réactions du logiciel.

Dans certaines des lignes de saisie ci-dessous, on trouvera l'instruction **restart**, qui permet de s'assurer que toutes les variables sont préalablement *désassignées* (c'est-à-dire n'ont plus de *contenu*).

QUESTION 1.1

```
> restart:  x:=y:  y:=1:  x;
```

QUESTION 1.2

```
> y:=1:  x:=y:  y:=2:  x;
```

QUESTION 1.3

```
> x:=1:  x:=x+1:  x;
```

QUESTION 1.4

```
> restart:  x:=x+1:  x;
```

QUESTION 1.5

```
> restart:  x:=2*y:  y=1:  x;
```

QUESTION 1.6

```
> restart:  x:=2*y:  y:=3*z:  u:=x+1:  x:=4:  u;
```

QUESTION 1.7

```
> restart:  x:=2*y:  y:=z+1:  z:=x+y:  u:=3*z;
```

QUESTION 1.8

```
> restart:  y:=10*x:  x:=1:  y; x:='x':  y;
```

QUESTION 1.9

```
> y:=4: x:=1+y: x-y; x:='1+y': x-y;
```

QUESTION 1.10

```
> x:=3: y:=x.x; y:=y.y;
```

QUESTION 1.11

```
> x:=' '+: x+x-2;
```

QUESTION 1.12

```
> x:='4 # Ici, on appuie sur Shift+Enter
x+1;
```

2.1.10. Réponses au test 1

RÉPONSE 1.1

La réponse de Maple est

1

restart réinitialise toutes les variables.

On met d'abord le nom y dans la variable x , puis la valeur 1 dans la variable y .

Dans la dernière instruction, la variable x s'évalue en y , qui s'évalue en 1.

RÉPONSE 1.2

La réponse de Maple est

1

On met 1 dans y , puis le contenu de y (donc 1) dans x .

On place ensuite 2 dans x , mais y reste inchangé.

La dernière instruction renvoie le contenu de x , c'est-à-dire 1.

RÉPONSE 1.3

La réponse de Maple est

2

La première instruction place 1 dans la variable x .

L'expression $x + 1$ s'évalue alors en 2, et ce résultat est placé dans x .

La dernière instruction se contente d'évaluer et d'afficher le contenu de x .

RÉPONSE 1.4

La réponse de Maple est

Warning, recursive definition of name

Error, too many levels of recursion

L'instruction `restart` vide toutes les variables, et notamment `x`.

L'expression $x + 1$ est ensuite placée dans la variable `x`.

Maple ne proteste pour l'instant que par un message de mise en garde (*warning*).

En revanche l'évaluation du contenu de `x` provoque une erreur (récursion infinie.)

En effet `x` s'évaluerait en $x + 1$, qui s'évaluerait ensuite en $(x + 1) + 1 = x + 2$, etc.

RÉPONSE 1.5

La réponse de Maple est

$$2y$$

On place l'expression $2y$ dans la variable `x`, puis 1 dans la variable `y`.

L'évaluation de `x` conduit donc à $2y$ puis finalement à $2 \times 1 = 2$, seul résultat affiché.

RÉPONSE 1.6

La réponse de Maple est

$$6z + 1$$

On place l'expression $2y$ dans la variable `x`, puis l'expression $3z$ dans la variable `y`.

L'expression $x + 1$ s'évalue alors en $2y + 1$ puis en $6z + 1$.

C'est ce résultat qui est placé dans la variable `u`.

Ensuite on place la valeur 4 dans la variable `x`.

Mais `u` contient toujours l'expression $6z + 1$, d'où le résultat.

RÉPONSE 1.7

La réponse de Maple est

Warning, recursive definition of name

Error, too many levels of recursion

On place $2y$ dans la variable `x` puis $z + 1$ dans la variable `y`.

L'expression $x + y$ s'évalue alors en $2y + z + 1$ puis en $3(z + 1)$.

Cette expression, dépendant de z , est ensuite placée dans la variable `z`.

C'est la raison du premier message d'avertissement.

Enfin l'évaluation de l'expression $3z$ conduit à une erreur de récursion infinie.

En effet $3z$ s'évalue en $9(z + 1)$, puis en $9(3(z + 1) + 1)$, etc.

RÉPONSE 1.8

La réponse de Maple est

$$10$$

$$10x$$

On place $10x$ dans **y** et 1 dans **x** ; **y** s'évalue alors en $10x = 10$.

Ensuite on vide la variable **x** de son contenu.

La variable **y** s'évalue toujours en $10x$, mais cela s'arrête là.

RÉPONSE 1.9

La réponse de Maple est

$$1$$

$$1 + y - 4$$

On met 4 dans la variable **y**, puis $1 + y$, et donc 5, dans la variable **x**.

Il est donc normal que $x - y$ s'évalue en $5 - 4 = 1$.

Ensuite on place le *nom* **1+y** dans la variable **x**.

$x - y$ s'évalue alors en '1y'-4, sauf que les caractères ' ne sont pas affichés.

RÉPONSE 1.10

La réponse de Maple est

$$y := x3$$

$$y := yx3$$

On met 3 dans **x**. Puis on forme le nom $x \cdot x$ par concaténation de deux noms x .

Le deuxième x s'évalue, mais pas le premier.

Le nom $x \cdot x$ s'évalue donc en le nom $x3$, et ce résultat va dans la variable **y**.

Par concaténation, le nom $y \cdot y$ s'évalue en $yx3$, suite à l'évaluation du deuxième y .

RÉPONSE 1.11

La réponse de Maple est

$$2 - 2$$

On met le *nom vide* dans la variable **x**.

L'expression $x + x - 2$ se simplifie en $2x - 2$ qui s'évalue en $2 - 2$.

C'est ce résultat qui est affiché, à l'exception des caractères '.

En tout cas le résultat final n'est pas la différence $2 - 2$ qui se simplifierait en 0.

RÉPONSE 1.12

La réponse de Maple est

Warning, incomplete quoted name; use ' to end the name

A la première ligne débute la création d'un nom débutant par le chiffre 4.

Le saut de ligne est considéré comme un caractère de ce nom.

Quand la deuxième ligne est validée, elle est traitée par Maple (caractère terminal ;).

A ce moment, Maple constate l'absence du second ' pour terminer le nom.

2.2. NOMBRES ET CONSTANTES

2.2.1. Nombres entiers

Maple effectue des calculs sur les nombres entiers relatifs de manière exacte, c'est-à-dire avec autant de chiffres que nécessaire.

Rappelons qu'un entier très long peut être découpé en tranches par des caractères \. Ceux-ci sont ensuite ignorés par Maple.

Les caractères \ sont utilisés par Maple lui-même pour afficher les entiers qui ne tiennent pas sur une seule ligne, à raison d'un caractère \ par saut de ligne.

2.2.2. Nombres rationnels

Les nombres rationnels sont les quotients $r = \frac{a}{b}$ de deux entiers relatifs, b étant non nul.

Quand un tel rationnel r est évalué, il est automatiquement simplifié en un entier si b divise a , et sinon en un rationnel $\frac{c}{d}$, c et d étant premiers entre eux, et d étant strictement positif.

Les calculs arithmétiques sur les rationnels, en particulier les mises au dénominateur commun, sont automatiquement effectués.

2.2.3. Nombres réels

Un nombre est au format réel s'il contient un point décimal, éventuellement à la fin : par exemple 5. est un nombre réel alors que 5 est un nombre entier.

Les calculs sur les nombres réels peuvent être effectués avec autant de décimales que nécessaire : cela dépend du contenu de la variable **Digits** (attention à la majuscule D), automatiquement créée par Maple et qui désigne précisément le nombre de décimales à utiliser. Par défaut, **Digits** contient 10. On peut le modifier à tout moment.

Dans une expression contenant à la fois des entiers (ou des rationnels) et des réels, tous les nombres sont convertis en nombres réels : on peut donc dire que les réels sont *contagieux*.

Une autre manière de créer un nombre réel est d'écrire `Float(mantisse,exposant)`, ce qui produit le nombre réel $\text{mantisse} * 10^{\text{exposant}}$.

2.2.4. Évaluation forcée au format réel

Il est toujours possible de forcer le passage en format réel d'une expression *expr* dont la valeur est entière ou rationnelle, ou qui contient des constantes symboliques comme **Pi** par exemple. On utilise pour cela l'instruction `evalf` (littéralement *eval to float*).

La syntaxe est `evalf(expr)` si on veut une évaluation avec **Digits** décimales, ou `evalf(expr,n)` si on veut une évaluation avec n décimales : n désigne ici un entier ou une expression s'évaluant sur un entier. Dans *expr*, tous les éléments qui ne peuvent être évalués en des nombres réels restent alors inchangés.

2.2.5. Nombres complexes

Maple nomme I (attention à la majuscule!) le complexe de module 1 et d'argument $\frac{\pi}{2}$.

Un nombre complexe est alors une expression $a + b * I$, où a et b sont réels ou rationnels.

Pour Maple, I n'est pas considéré comme une constante, mais comme un synonyme de $\sqrt{-1}$.

L'expression I^2 est automatiquement simplifiée en sa valeur -1 .

L'instruction `evalc` (littéralement *eval to complex*) peut être utilisée pour évaluer une expression *expr* dont on sait qu'elle représente un nombre complexe, en l'écrivant sous forme cartésienne $a + I * b$, où a et b sont des expressions à valeurs réelles.

Le point essentiel est que les noms non assignés qui figureraient dans *expr* seront considérés par `evalc` comme ayant un contenu réel.

2.2.6. Constantes symboliques

Maple définit lui-même un certain nombre de constantes.

On trouve leur *séquence* dans la variable globale `constants` :

```
> lprint(constants); constants; evalf(%);
false gamma infinity true Catalan FAIL Pi
```

false, γ , ∞ , true, Catalan, FAIL, π

false, .5772156649, ∞ , true, .9159655942, FAIL, 3.141592654

On peut définir de nouvelles constantes symboliques en ajoutant leurs noms à la séquence contenue dans la variable `constants`.

Ces noms peuvent être déjà assignés ou non. Dans tous les cas, on peut ensuite modifier le contenu affecté à ces noms, même si ce n'est pas très logique pour des constantes!

```
> j:=1: k:='k': constants:=constants,'j',k; j:=2;
```

constants := false, γ , ∞ , true, Catalan, FAIL, π , j, k

j := 2

2.2.7. Expressions constantes

Une expression *expr* est de type *constant* si sa valeur s'écrit à partir de constantes symboliques ou de nombres, par des opérations de sommation, produit, quotient, exponentiation, ou en utilisant des fonctions dont les arguments sont constants.

Si par exemple le résultat de *expr* après évaluation contient un nom qui n'est pas affecté et qui n'a pas été déclaré comme constante symbolique, alors *expr* n'est pas de type constant.

Même si une expression n'est pas de type constant, il arrive fréquemment que certaines de ses sous-expressions soient de ce type.

2.2.8. Fonctions avec arguments constants

L'appel à une fonction *func* nécessitant n arguments s'écrit $func(arg_1, arg_2, \dots, arg_n)$. Cette expression peut à elle seule constituer une instruction, ou elle peut être le membre droit d'une assignation $name := func(\dots)$.

Dans les deux cas cette expression doit être évaluée.

Pour cela il faut évaluer chacun des arguments (on aboutit à leurs valeurs val_1, val_2 , etc.), puis calculer si possible l'image $func(val_1, val_2, \dots, val_n)$.

Ce schéma très général recouvre des mécanismes complexes et il y a des exceptions.

Plaons nous par exemple dans le cas d'une fonction numérique usuelle *func* qui ne nécessite qu'un seul argument, comme la fonction **sin** (*sinus*) ou la fonction **sqrt** (*racine carrée*). Si l'argument *arg* de la fonction *func* est une constante, ou plutôt s'il a une valeur *val* constante, tout dépend de la nature de cette constante.

Si cette valeur *val* est un nombre au format réel, alors la valeur $func(val)$ est effectivement calculée de manière approchée avec la précision définie par **Digits**, à moins qu'on ne modifie cette précision avec la syntaxe **evalf**($func(arg), n$), où n est un entier.

Si *val* est une constante ne contenant pas de nombres réels mais des entiers, des rationnels, des constantes symboliques, ou des complexes $a + I * b$ avec les mêmes conditions sur a et b , alors $func(val)$ n'est pas évalué à moins que le résultat ne soit plus *simple* que l'original.

Cette notion de simplicité est vague mais concerne surtout ici les valeurs classiques comme $\exp 0$, $\ln 1$, $\sin \frac{\pi}{3}$, les racines carrées d'entiers, etc. Certaines propriétés (périodicité, parité ou imparité, etc.) des fonctions usuelles seront donc automatiquement utilisées dans ce processus de simplification.

Il est toujours possible de forcer l'évaluation vers le format réel en utilisant l'instruction **evalf** avec un deuxième argument éventuel n pour demander n décimales.

2.2.9. Test 2

Imaginez les réponses à chacune des lignes de saisie suivantes. Comparer ensuite avec ce que répond effectivement Maple, et justifiez les réactions du logiciel.

Dans certaines des lignes de saisie ci-dessous, on trouvera l'instruction **restart**, qui permet de s'assurer que toutes les variables sont préalablement *désassignées* c'est-à-dire n'ont plus de *contenu*, ou encore que la variable globale **Digits** contient bien 10, sa valeur par défaut.

Rappel: si $expr_1, expr_2, \dots, expr_n$ sont des expressions, l'instruction :

$$expr_1, expr_2, \dots, expr_n;$$

provoque un affichage des valeurs de ces expressions sur une même ligne (si possible), alors que la ligne de saisie

$$expr_1; expr_2; \dots; expr_n;$$

affiche ces valeurs à raison d'un résultat par ligne.

QUESTION 2.1

```
> 10^100;
```

QUESTION 2.2

```
> Float(1,100), 10^100-Float(1,100);
```

QUESTION 2.3

```
> 1+2/10+3/100+4/1000;
```

QUESTION 2.4

```
> 1/5+5-5.0, 1/5+0.0, 1/5*1.;
```

QUESTION 2.5

```
> restart: x:=1/3: y:=evalf(%): y, evalf(%,15), evalf(x,15);
```

QUESTION 2.6

```
> restart: (a+I)^2, (1+I)^2, exp(I*Pi/2);
```

QUESTION 2.7

```
> restart: exp(1+I*alpha): %, evalc(%);
```

QUESTION 2.8

```
> restart: cos(pi): %, evalf(%), sigma*psi*theta
```

QUESTION 2.9

```
> restart: cos(-x), sin(-x), cos(x+37*Pi), cos(x+Pi/2);
```

QUESTION 2.10

```
> x:=(sqrt(2)-1)^2: x, expand(x), evalf(x);
```

QUESTION 2.11

```
> cos(Pi/2), cos(Pi^2), exp(1), exp(2), ln(1), ln(3^2);
```

QUESTION 2.12

```
> ln(e), ln(exp(1)), evalf(e), evalf(exp(1));
```

2.2.10. Réponses au test 2

RÉPONSE 2.1

La réponse de Maple est

[illegible]

RÉPONSE 2.2

La réponse de Maple est

$$.1\,10^{101}, 0$$

Float(1,100) désigne 10^{100} , mais au format réel.

Dans `10^100-Float(1,100)`, l'entier 10^{100} est converti au format réel.

La différence est donc 0 de manière exacte.

RÉPONSE 2.3

La réponse de Maple est

$$\frac{617}{500}$$

Le calcul donne le rationnel $1,234 = \frac{1234}{1000}$, automatiquement simplifié en $\frac{617}{500}$.

RÉPONSE 2.4

La réponse de Maple est

$$.2000000000, \frac{1}{5}, .2000000000$$

Le premier résultat est $\frac{1}{5}$, converti au format réel à cause de la présence du réel 5.0

Dans l'expression suivante, le réel 0.0 est automatiquement éliminé.

C'est pourquoi le résultat est encore $\frac{1}{5}$ au format exact.

Dans la troisième expression, le produit par le réel 1.0 force le passage au mode réel.

C'est pourquoi le troisième résultat est le réel 0.2.

RÉPONSE 2.5

La réponse de Maple est

.3333333333, .3333333333, .333333333333333

On met le rationnel $\frac{1}{3}$ dans la variable `x`, et aussi dans `%`.

`evalf(%)` s'évalue en la forme réelle de $\frac{1}{3}$, avec les 10 décimales par défaut.

C'est ce résultat qui va alors dans la variable `y` et dans `%`.

Il est alors normal que l'évaluation de `y` renvoie .3333333333

`evalf(%, 15)` voudrait exprimer avec 15 décimales un réel qui n'en contient que 10.

Il est donc normal que `evalf(%, 15)` renvoie encore .3333333333

En revanche `evalf(x, 15)` évalue le contenu de `x`, donc $\frac{1}{3}$, avec 15 décimales.

Le résultat est donc .333333333333333 (compter le nombre de 3).

RÉPONSE 2.6

La réponse de Maple est

$$(a + I)^2, 2I, I$$

$(a + i)^2$ n'est pas automatiquement simplifié car on ne sait rien sur le contenu de `a`.

Mais $(1 + i)^2$ est évalué et simplifié en $2i$.

De même, Maple simplifie $\exp \frac{i\pi}{2}$ en i de manière automatique.

RÉPONSE 2.7

La réponse de Maple est

$$e^{(1+I\alpha)}, e \cos(\alpha) + I e \sin(\alpha)$$

Maple laisse inchangée l'expression $\exp(1 + i\alpha)$, car on ne sait rien de α .

En revanche, l'instruction `evalc` présuppose que le contenu de α est réel.

Le résultat est alors développé sous forme cartésienne $a + ib$.

RÉPONSE 2.8

La réponse de Maple est

$$\cos(\pi), \cos(\pi), \sigma \psi \text{ teta}$$

Pour Maple la constante mathématique π s'écrit `Pi`.

Elle ne s'écrit donc ni `PI`, ni `pi`.

C'est pourquoi Maple refuse de simplifier ou d'évaluer numériquement `cos(pi)`.

Ce qui rend la faute difficile à voir, c'est que `pi` et `Pi` s'affichent tous deux en π .

Enfin, on voit comment Maple reconnaît et affiche *en grec* les lettres σ et ψ .

En revanche, la lettre grecque θ (*theta*), mal orthographiée, n'est pas reconnue.

RÉPONSE 2.9

La réponse de Maple est

$$\cos(x), -\sin(x), -\cos(x), -\sin(x)$$

Maple sait que les fonctions \cos et \sin sont l'une paire et l'autre impaire.

Il sait aussi qu'elles sont 2π -périodiques.

Il connaît aussi la formule $\cos(x + \frac{\pi}{2}) = -\sin x$ (entre autres...).

Cela explique que Maple ait automatiquement simplifié ces expressions.

RÉPONSE 2.10

La réponse de Maple est

$$(\sqrt{2} - 1)^2, 3 - 2\sqrt{2}, .1715728749$$

On sait que par exemple $(\mathbf{I} - 1)^2$ est automatiquement développé.

On voit qu'il n'en est rien avec $(\sqrt{2} - 1)^2$. Il suffit de le savoir.

RÉPONSE 2.11

La réponse de Maple est

$$0, \cos(\pi^2), e, e^2, 0, \ln(9)$$

RÉPONSE 2.12

La réponse de Maple est

$$\ln(e), 1, e, 2.718281828$$

On voit que pour Maple la lettre **e** ne désigne pas la constante mathématique homonyme.

Pour y faire référence, il faut utiliser l'expression $\exp(1)$.

2.3. OPÉRATEURS

2.3.1. Types d'opérateurs

Un *opérateur* représente une *opération* à effectuer sur un ou deux *arguments*.

On distingue les opérateurs *unaires* (un seul argument) des opérateurs *binaires* (deux arguments). Les opérateurs unaires seront dits *préfixés* ou *postfixés* selon qu'il précèdent ou suivent leur argument.

Les opérateurs binaires seront dits *infixés* car ils séparent leurs deux arguments.

Nous considérerons surtout les opérateurs *arithmétiques*, *logiques*, ou de *relation*.

2.3.2. Opérateurs arithmétiques

Ils sont très utilisés, car ils servent à former toutes sortes d'expressions algébriques.

Dans la catégorie *unaire*, on trouve :

$\boxed{+}$: préfixé, mais sans effet. Il disparaît à l'affichage.

$\boxed{-}$: préfixé. C'est le changement de signe. Ne pas le confondre avec la soustraction.

$\boxed{!}$: postfixé. C'est la factorielle. Par exemple $3!$ vaut $3 * 2 = 6$ et $3!!$ vaut $6! = 720$.

L'argument de la factorielle ne doit pas être négatif.

Dans la catégorie *binaire*, on trouve :

$\boxed{+}$: c'est l'addition, toujours commutative et associative.

On peut considérer des sommes $a + b + c + \dots$ sans risque d'ambiguïtés.

Quand on évalue une somme, les termes semblables sont automatiquement regroupés.

Si le résultat final est encore une somme, on ne peut pas prévoir l'ordre des termes.

$\boxed{-}$: c'est la soustraction.

En interne, $a - b$ est codé $a + (-b)$ (somme de a et de l'opposé de b).

$\boxed{*}$: c'est le produit, toujours commutatif et associatif.

On peut donc considérer des produits $a * b * c * \dots$ sans risque d'ambiguïtés.

Quand on évalue un produit, les termes semblables sont automatiquement regroupés.

Si le résultat final est encore un produit, on ne peut pas prévoir l'ordre des termes.

$\boxed{/}$: c'est le quotient.

En interne, $\frac{a}{b}$ est codé $a * b^{-1}$ (produit de a par l'inverse de b).

$\boxed{\wedge}$: c'est l'exponentiation. On peut utiliser le synonyme $**$.

Cet opérateur n'est pas associatif. On écrira $a \wedge (b \wedge c)$ ou $(a \wedge b) \wedge c$, mais pas $a \wedge b \wedge c$.

Pour obtenir le caractère \wedge à l'écran, sans risque d'obtenir un des caractères accentués, on fera suivre l'appui sur la touche \wedge par un appui sur la *barre d'espace*.

`mod` : c'est le *modulo*.

$a \bmod b$ calcule le reste dans la division de l'entier a par l'entier non nul b .

`&*` : c'est le *produit inerte*.

La commutativité de l'opérateur `*` est normale quand les arguments sont des nombres par exemple, mais elle peut conduire à des erreurs si on l'applique à des objets pour lesquels le produit usuel n'est pas commutatif, et en particulier pour les matrices.

On aura alors recours à l'opérateur `&*` qui est une sorte de produit *inerte*.

Dans un contexte matriciel, on utilisera l'instruction `evalm` (littéralement *evaluate to matrix*) pour forcer l'évaluation de ces produits.

`&^` : c'est l'*exponentiation inerte*.

Par exemple, $a \&^b \bmod m$ calcule le reste dans la division de a^b par m , mais sans calculer a^b , ce que ferait $a^b \bmod m$.

2.3.3. Opérateurs logiques

Maple connaît trois constantes symboliques logiques :

`true` représente la valeur logique *vrai*.

`false` représente la valeur logique *faux*.

`FAIL` est une valeur logique qu'on pourrait désigner par *je ne sais pas*.

Elle résulte souvent d'un échec de Maple à dire si une condition est réalisée ou non.

Maple utilise donc une logique *trivaluée*.

Les opérateurs logiques agissent sur des expressions dont la valeur est *booléenne*, c'est-à-dire l'une des trois constantes symboliques rappelées précédemment.

Il y a un opérateur logique unaire qui est `not` (négation logique). Les opérateurs logiques binaires sont `and` (*et* logique) et `or` (*ou* logique). On remarque que *xor* (*ou* exclusif) est absent, mais le package `logic` introduit toute une panoplie de nouveaux opérateurs.

`not` est un opérateur préfixé : on utilisera la syntaxe `not(expr)` ou `notexpr`.

Voici comment se comportent les opérateurs logiques sur la constante `FAIL`:

```
> FAIL and true, FAIL and false, FAIL and FAIL;
```

FAIL, false, FAIL

```
> FAIL or true, FAIL or false, FAIL or FAIL, not FAIL;
```

true, FAIL, FAIL, FAIL

2.3.4. Opérateurs de relation

Les opérateurs de relation servent à former des égalités ou des inégalités.

Ce sont $\left\{ \begin{array}{ll} < \text{ strictement inférieur} & \leq \text{ inférieur ou égal} \\ > \text{ strictement supérieur} & \geq \text{ supérieur ou égal} \\ = \text{ égal} & \neq \text{ différent} \end{array} \right.$

Ce sont tous des opérateurs binaires.

Une expression *expr* représentant une égalité ou une inégalité ne sera *testée* (c'est-à-dire évaluée vers l'une des constantes **true**, **false**, ou **FAIL**) que dans un *contexte booléen*, c'est-à-dire en général dans une structure du type *if expr then...*, ou *while expr do...*

Dans un *contexte algébrique*, les opérateurs de relation servent à former des *équations* ou des *inéquations*, qui pourront ensuite être *résolues*, par exemple par **solve**.

Il est toujours possible de forcer l'évaluation booléenne d'une expression contenant des opérateurs de relation en utilisant l'opérateur **evalb** (littéralement *evaluate to boolean*).

2.3.5. Autres types d'opérateurs

Sans rentrer dans les détails, il existe de nombreux autres opérateurs.

Rappelons seulement les trois opérateurs binaires suivants :

:= C'est l'*assignation*. Par exemple **x:=1** place 1 dans la variable **x**.

. Le *point* permet de former des noms par concaténation.

, La *virgule* sert de séparateur entre deux expressions.

2.3.6. Questions de priorité

Quand une expression *expr* contenant des opérateurs est évaluée, certains opérateurs sont mis en oeuvre avant d'autres, selon des règles classiques de priorité (*précédence*).

Par exemple le produit ***** est toujours prioritaire devant l'addition.

On peut modifier l'ordre d'évaluation en utilisant des parenthèses (leur contenu est toujours traité en priorité). L'utilisation de parenthèses est recommandée quand on a un doute.

Voici l'ordre des priorités décroissantes, pour les opérateurs déjà rencontrés :

point, factorielle, exponentiation, produit et quotient, somme et différence, modulo, relations, opérateurs logiques (dans l'ordre **not**, **and**, **or**), *virgule, assignation*.

Par exemple, dans l'instruction *(name := a + b * c!, d)* :

- La quantité $f = c!$ est d'abord évaluée.
- Ensuite c'est le tour du produit $p = b * f$.
- La somme $s = a + p$ est alors calculée.
- A ce moment, Maple forme la séquence $seq = s, d$.
- Enfin cette séquence *seq* est affectée à la variable *name*.

2.3.7. Test 3

Imaginez les réponses à chacune des lignes de saisie suivantes.

Comparer ensuite avec les réponses données par Maple, et justifiez les réactions du logiciel.

Dans certaines des lignes de saisie, on trouvera l'instruction **restart**, qui permet de s'assurer que toutes les variables sont préalablement *désassignées* (c'est-à-dire n'ont plus de *contenu*).

Rappel: si $expr_1, expr_2, \dots, expr_n$ sont des expressions, l'instruction :

$$expr_1, expr_2, \dots, expr_n;$$

affiche les valeurs de ces expressions sur une même ligne (si possible), alors que :

$$expr_1; expr_2; \dots; expr_n;$$

affiche ces valeurs à raison d'un résultat par ligne.

QUESTION 3.1

```
> restart:  x*y*x*y-y*x*y*x, x**y-y**x, x&*y-y&*x;
```

QUESTION 3.2

```
> restart:  x:=1:  x+'x', 'x+x', y.(1-x)+xy-x(y+1);
```

QUESTION 3.3

```
> 2^3!, 7+ 5 mod 3;
```

QUESTION 3.5

```
> restart; not true and x, not (true and x), false or x;
```

QUESTION 3.6

```
> true or false and false, (true or false) and false;
```

QUESTION 3.7

```
> FAIL or true, FAIL and true, not FAIL or FAIL,
  not (FAIL or FAIL);
```

QUESTION 3.8

```
> 2<3, evalb(2<3), 2<3 and 3<4,
  exp(3)<exp(4), evalb(exp(3)<exp(4));
```

QUESTION 3.9

```
> restart:  x=y, evalb(x=y);
  x:=1:  y:=2:  x=y, evalb(x=y);
```


QUESTION 3.10

```
> restart:  x<>y and x=y, x<=y and x<>y, x<y or x=y;
```

QUESTION 3.11

```
> true-true, false-false, FAIL-FAIL, true+false,
   true+true,true+FAIL;
```

2.3.8. Réponses au test 3

RÉPONSE 3.1

La réponse de Maple est

$$0, x^y - y^x, (x \&* y) - (y \&* x)$$

$xyxy - yxyx$ se simplifie en 0, par associativité et commutativité.

L'opérateur $*$ équivaut à l'exponentiation.

$x\&*y - y\&*x$ reste inchangée car le produit inerte $\&*$ n'est pas supposé commutatif.

RÉPONSE 3.2

La réponse de Maple est

$$2, x + x, y0 + xy - 1$$

On commence par mettre la valeur 1 dans la variable \mathbf{x} .

Dans l'expression $x + 'x'$, $'x'$ désigne le nom x .

Il est donc normal que $x + 'x'$ s'évalue en $2x$ puis en 2.

L'expression $'x + x'$ désigne le *nom* $x + x$, qui ici ne contient rien.

$y.(1 - x)$ est un nom, concaténation de y (non évalué) et $1 - x$ (évalué).

En tout cas $y.(1 - x)$ ne désigne pas le produit de y par $(1 - x)$!

De même, xy désigne un nom (ici sans contenu) et pas le produit de x par y .

Enfin $x(y + 1)$ est l'image en $y + 1$ de la fonction x , qui est ici constante de valeur 1.

RÉPONSE 3.3

La réponse de Maple est

$$64, 0$$

$2^3!$ s'évalue comme $2^{(3!)}$, car la factorielle est prioritaire sur le produit.

$7 + 5 \bmod 3$ s'évalue comme $(7 + 5) \bmod 3$ car $+$ est prioritaire sur \bmod .

QUESTION 3.4

```
> restart:  x.3, x3, x*3, x(3);
           x:=5:  x.3, x3, x*3, x(3);
```

RÉPONSE 3.4

La réponse de Maple est

$$x^3, x^3, 3x, x(3)$$

$$x^3, x^3, 15, 5$$

$x.3$ et $x3$ désigne tous les deux le même nom.

L'expression $x * 3$, automatiquement réécrite en $3 * x$, est le produit de x par 3.

L'expression $x(3)$ désigne la valeur au point 3 de la fonction nommée x .

Ce n'est que dans ces deux derniers cas que le nom x est évalué.

On le voit bien en comparant les deux lignes.

RÉPONSE 3.5

La réponse de Maple est

$$false, \text{ not } x, x$$

Maple applique ici des règles de simplification automatique.

`not true` est converti en `false`, et `false and x` simplifié en `false`.

`true and x` est simplifié en `x` ; mais `not` ne sait rien de `x`.

On voit ici l'importance des parenthèses (`not` est prioritaire sur `and`).

Enfin `false or x` se simplifie automatiquement en `x`.

Donc `false` est *neutre* pour l'opérateur `or`, et `true` est neutre pour `and`.

RÉPONSE 3.6

La réponse de Maple est

$$true, false$$

Sans parenthèses, `or` est évalué en premier, et `true or...` s'évalue en `true`.

Dans le deuxième cas, `true or false` se simplifie en `true`.

Ensuite `true and false` se simplifie en `false`.

RÉPONSE 3.7

La réponse de Maple est

$$true, FAIL, FAIL, FAIL$$

Comme on le voit, l'usage de la constante `FAIL` nécessite de la prudence.

De plus, on ne confondra pas la constante `FAIL` avec le nom `fail`.

RÉPONSE 3.8

La réponse de Maple est

$$2 < 3, \text{true}, \text{true}, e^3 < e^4, e^3 - e^4 < 0$$

On voit sur cet exemple que les conditions dans lesquelles sont évalués les opérateurs relationnels ne sont pas toujours très claires.

Ces problèmes conduisent à des erreurs de programmation difficiles à déceler.

Remarque : `evalb((evalf(exp(3)<exp(4)))` s'évalue bien en `true`.

RÉPONSE 3.9

La réponse de Maple est

$$x = y, \text{false}$$

$$1 = 2, \text{false}$$

En matière de calcul symbolique, Maple choisit souvent le *cas général*.

Sans rien connaître de `x` et de `y`, l'évaluation de `x=y` donne donc `false`.

Quand `x` et `y` ont un contenu, l'expression `x=y` désigne encore une égalité.

En l'occurrence il s'agit de `1=2`, qui n'est évaluée en `false` que si on le demande.

RÉPONSE 3.10

La réponse de Maple est

$$\text{false}, x - y \leq 0, x - y < 0$$

Ces réponses se comprennent quand on sait que, lorsque `x` et `y` sont sans contenu, l'évaluation booléenne de `x=y` donne `false`, et celle de `x<>y` donne `true`.

On aurait sans doute préféré que Maple simplifie la troisième expression en `x<=y`.

RÉPONSE 3.11

La réponse de Maple est

$$0, 0, \text{FAIL}, \text{true} + \text{false}, 2 \text{true}, \text{FAIL}$$

Les réponses obtenues ici sont plus ou moins prévisibles.

Mais où serait l'intérêt d'évaluer de telles expressions?

2.4. FONCTIONS

2.4.1. Les fonctions dans Maple

Il n'est pas évident de délimiter précisément la notion de fonction dans un logiciel de calcul symbolique. Disons simplement qu'une fonction de Maple est un objet f qui agit sur des *arguments* pour *renvoyer* un *résultat*.

L'expression $f(arg_1, \dots, arg_n)$ représente alors (avant évaluation) l'image par la fonction f de la séquence d'arguments arg_1, \dots, arg_n .

La définition précédente est vague et on peut se demander si tout n'est pas fonction dans Maple. Les opérateurs (unaires ou binaires) peuvent ainsi être considérés comme des fonctions s'appliquant à un ou à deux arguments, avec simplement une syntaxe d'appel un peu particulière. Même un nombre peut être considéré comme une fonction constante.

```
> restart: x:=2: x(y+1), x(j,k), x(4,5,6,7), x();
```

2, 2, 2, 2

Dans toute la suite nous considérerons des fonctions au sens habituel du terme, comme les fonctions mathématiques usuelles connues de Maple, ou des fonctions qui pourraient être définies par l'utilisateur.

2.4.2. Opérations sur les fonctions

Si f est une fonction, on ne confondra pas l'objet f , et l'objet $f(args)$ (où $args$ est une séquence d'arguments éventuellement vide) qui est une expression.

On peut assigner une fonction à un nom :

```
> f:=sin; f(Pi/2), f(3*Pi), f(Pi/3);
```

$1, 0, \frac{1}{2}\sqrt{3}$

On peut effectuer des opérations algébriques sur les fonctions :

```
> f:=3*sqrt-ln^2; f(5), f(1);
```

$f := 3\sqrt{} - \ln^2$
 $3\sqrt{5} - \ln(5)^2, 3$

L'opérateur @ (*arrobas*) permet de composer des fonctions :

```
> restart: f:=sqrt@ln: g:=ln@sqrt: f(x), g(x), f@g(x), (f@g)(x);
```

$\sqrt{\ln(x)}, \ln(\sqrt{x}), \sqrt{\ln(\ln(\sqrt{x}))}, \sqrt{\ln(\ln(\sqrt{x}))}$

On remarquera que dans $f@g(x)$, la sous-expression $g(x)$ est évaluée avant l'opérateur @. Il faudra donc placer $f@g$ entre parenthèses si on veut calculer $f_{\circ}g(x)$.

Maple simplifie automatiquement la composée d'une fonction et de son inverse :

```
> (sin@arcsin)(x), exp(ln(x)), (exp@ln)(x), exp@ln(x);
```

$$x, x, x, \exp@(\ln(x))$$

On peut utiliser la syntaxe $f@@n$ pour former la composée $f \circ f \circ \dots \circ f$ (n fois) :

```
> f:=sqrt@@4: f(5), (sqrt@@4)(5), sqrt@@4(5);
```

$$5^{1/16}, 5^{1/16}, \text{sqrt}^{(4)}$$

On remarquera dans le dernier exemple ci-dessus, que l'expression $4(5)$ est d'abord évaluée (ce qui conduit à la valeur 4) et qu'ensuite l'expression $\text{sqrt}@@4$ est évaluée. Le résultat est une fonction dans laquelle l'*exposant* (4 entre parenthèses) ne doit pas être interprété comme un ordre de dérivation, mais comme un ordre de composition !

La syntaxe $f@@(-1)$ permet dans certains cas de former la fonction inverse de f (encore faut-il qu'elle existe et que Maple puisse la trouver) :

```
> sin@@(-1), arcsin@@(-1), ln@@(-1), ln@@(-3);
```

$$\arcsin, \sin, \exp, \exp^{(3)}$$

Parmi tous les opérateurs, la précedence de @@ est la même que celle de l'opérateur $^$ d'exponentiation, alors que l'opérateur @ a même priorité que le produit $*$ ou le quotient $/$.

D permet de dériver une fonction (le résultat est encore une fonction).

```
> D(sin), D(cos+sqrt), (D@@6)(sin);
```

$$\cos, -\sin + \frac{1}{2} \frac{1}{\text{sqrt}}, -\sin$$

2.4.3. Définir des fonctions

Le nombre de fonctions intégrées à Maple est important, mais on a souvent besoin de définir, au moins momentanément, ses propres fonctions.

La syntaxe la plus simple pour cela est :

- Pour une fonction d'une seule variable, $f := \text{var} \rightarrow \text{expr}$.
- Pour une fonction de n variables, $f := (\text{var}_1, \dots, \text{var}_n) \rightarrow \text{expr}$.

Dans ces notations, f désigne le nom qu'on va donner à la fonction, var le nom de sa variable (ou $\text{var}_1, \dots, \text{var}_n$ le nom de ses variables, dans un ordre bien précis) et expr est une expression dépendant en général de var (respectivement de $\text{var}_1, \dots, \text{var}_n$).

```
> f:=x->x^2+sqrt(x); f(4), f(5);
```

$$f := x \rightarrow x^2 + \sqrt{x}$$

$$18, 25 + \sqrt{5}$$

```
> f:=(x,y)->x^3-y^3; f(10,2), f(a,b), f(b,a);
```

$$f := (x, y) \rightarrow x^3 - y^3$$

$$992, a^3 - b^3, b^3 - a^3$$

On peut très bien définir une fonction *identité* :

```
> restart: id:=x->x: (id@f)(x), (f@id)(x), (id@@5)(x);
```

$$f(x), f(x), x$$

Il n'est pas nécessaire de donner un nom à la fonction créée avec l'opérateur \rightarrow :

```
> x->x+x^2+x^3:%(1),%(2),%(a+b),(t->t+exp(t))(4);
```

$$3, 14, a + b + (a + b)^2 + (a + b)^3, 4 + e^4$$

2.4.4. L'instruction `unapply`

Les exemples qui précèdent montrent comment définir une fonction puis appliquer cette fonction à un ou plusieurs arguments, pour obtenir des expressions.

L'opération inverse (construire une fonction à partir d'une expression et connaissant le nom de la ou des variables) est également utile.

On utilise pour cela `unapply`. La syntaxe est `unapply(expr, var)` dans le cas d'une variable ou `unapply(expr, var1, ..., varn)` dans le cas de n variables.

Le résultat est la fonction dont l'image (avec l'argument var ou les arguments var_1, \dots, var_n) serait l'expression $expr$.

```
> f:=unapply((3+x)/(1+x+x^2),x): f(100), f(a);
```

$$\frac{103}{10101}, \frac{3+a}{1+a+a^2}$$

```
> f:=unapply(sqrt(x+y)*exp(x*y),x,y): f(a,b), f(2,6);
```

$$\sqrt{a+b}e^{(ab)}, \sqrt{8}e^{12}$$

2.4.5. Test 4

Imaginez les réponses à chacune des lignes de saisie suivantes.

Comparer ensuite avec les réponses données par Maple, et justifiez les réactions du logiciel.

Dans certaines des lignes de saisie, on trouvera l'instruction **restart**, qui permet de s'assurer que toutes les variables sont *désassignées* (c'est-à-dire n'ont plus de *contenu*).

Rappel: si $expr_1, expr_2, \dots, expr_n$ sont des expressions, l'instruction

$$expr_1, expr_2, \dots, expr_n;$$

affiche les valeurs de ces expressions sur une même ligne (si possible), alors que

$$expr_1; expr_2; \dots; expr_n;$$

affiche ces valeurs à raison d'un résultat par ligne.

QUESTION 4.1

```
> restart:  f:=x->3(x+1)^2-5; f(a);
```

QUESTION 4.2

```
> restart:  x:=1:  f:=x->sin(x)+cos(x):  f(a),f(x);
```

QUESTION 4.3

```
> restart:  (f@@4)(f@@5), (f@@(-1))(f)(x);
```

QUESTION 4.4

```
> restart:  f:=x,y->sin(x)*cos(y); f(a,b);
```

QUESTION 4.5

```
> restart:  f:=(x,y)->sin(x)*cos(y);
f(a,b), f(b,a), f(y,x), f(x+y,x-y);
```

QUESTION 4.6

```
> restart:  f:=(x,y)->sin(x),cos(y); f(a,b);
```

QUESTION 4.7

```
> restart:  a(x+1), (a+1)((b+2)(c+3)), (x->x)(a);
```

QUESTION 4.8

```
> restart:  g:=x->exp(x):  h:=x->2*x+1:
(g@@(-1))(x), (h@@(-1))(x);
```

QUESTION 4.9

```
> restart:  g:=exp(x):  D(g), (g@g)(x), (g@@(-1))(x);
```

2.4.6. Réponses au test 4

RÉPONSE 4.1

La réponse de Maple est

$$f := x \rightarrow 3(x+1)^2 - 5$$

4

Il manque évidemment le symbole du produit entre 3 et $(x+1)$.

Ici Maple pense que $3(x+1)$ est la valeur en $x+1$ de la fonction constante égale à 3.

Il évalue donc $f(a)$ en $3(a+1)^2 - 5$, puis en $3^2 - 5$, et donc 4.

RÉPONSE 4.2

La réponse de Maple est

$$\sin(a) + \cos(a), \sin(1) + \cos(1)$$

Dans la définition de f , on utilise la variable \mathbf{x} .

Mais on l'utilise comme une variable muette, donc sans tenir compte de son contenu.

C'est pourquoi l'instruction $\mathbf{x}:=1$ n'influe pas sur la définition de f .

En revanche, quand on calcule $f(x)$, la variable \mathbf{x} est évaluée : on calcule donc $f(1)$.

RÉPONSE 4.3

La réponse de Maple est

$$f^{(9)}, x$$

On voit ici deux exemples de simplifications automatiques.

RÉPONSE 4.4

La réponse de Maple est

$$f := x, y \rightarrow \sin(x) \cos(y)$$

$$\mathbf{x}(a, b), \sin(x) \cos(a)$$

Il y a une erreur de syntaxe. On aurait dû écrire $\mathbf{f}:=(\mathbf{x},\mathbf{y})\rightarrow\sin(\mathbf{x})*\cos(\mathbf{y})$.

Maple comprend ici que f est la séquence x, g , si on appelle g la fonction qui à y associe $\sin x \cos y$.

Quand on évalue $f(a, b)$, Maple répond donc par la séquence $x(a, b), g(a, b)$.

g étant une fonction d'une seule variable $g(a, b)$ équivaut à $g(a)$.

Le résultat est donc $x(a, b), g(a)$, c'est-à-dire : $x(a, b), \sin x \cos a$.

RÉPONSE 4.5

La réponse de Maple est

$$f := (x, y) \rightarrow \sin(x) \cos(y) \\ \sin(a) \cos(b), \sin(b) \cos(a), \sin(y) \cos(x), \sin(x+y) \cos(x-y)$$

On doit retenir que les noms x et y utilisés dans la définition de f sont *muets*.

Poser `f:=(y,x)->sin(y)*cos(x)` par exemple est complètement équivalent.

RÉPONSE 4.6

La réponse de Maple est

$$f := (x, y) \rightarrow \sin(x), \cos(y) \\ \sin(a), \cos(y)(a, b)$$

Encore une faute de syntaxe... Ici Maple comprend que f est la séquence composée :

- De la fonction g qui serait définie par $g(x, y) = \sin x$.
- De l'expression h qui serait égale à $\cos y$.

Quand on évalue $f(a, b)$, Maple répond alors par $g(a, b), h(a, b)$.

Il aurait fallu écrire `f:=(x,y)->(sin(x),cos(y))`.

$f(a, b)$ se serait alors évalué en la séquence : $\sin a, \cos b$.

RÉPONSE 4.7

La réponse de Maple est

$$a(x+1), a(b(c+3)+2)+1, a$$

Dans les trois premières expressions, Maple pense que a et b sont des fonctions.

Dans la troisième expression, $x \rightarrow x$ est l'application identité.

RÉPONSE 4.8

La réponse de Maple est

$$\ln(x), (h^{(-1)})(x)$$

D'abord Maple remplace $\exp^{-1}(x)$ par $\ln(x)$. C'est une simplification automatique.

Ensuite, la bijection réciproque de h existe, mais Maple ne fait la substitution.

RÉPONSE 4.9

La réponse de Maple est

$$D(e^x), ((e^x)^{(2)})(x), ((e^x)^{(-1)})(x)$$

Encore une erreur assez courante...

En posant `g:=exp(x)` on définit une expression, et non pas une fonction.

Les opérateurs D (dérivation) et \circ (composition) n'agissent que sur des fonctions.

Il aurait fallu poser ici `g:=x->exp(x)`, ou mieux `g:=exp`.

2.5. EXPRESSIONS

2.5.1. Arborescence d'une expression

La plupart des objets de Maple sont des *expressions*.

Une expression est formée en appliquant des *fonctions* ou des *opérateurs* à des *arguments* (ou *opérandes*), ces opérandes pouvant eux-mêmes être des expressions formées avec des opérateurs, des fonctions, et... des opérandes.

Cette définition est donc récursive. Mais on voit qu'il y a, dans une expression donnée, des éléments qui sont à la *surface* et d'autres qui sont plus *internes*.

Par exemple, dans l'expression $\exp(x^a + y)$, la fonction **exp** est à la tête de l'expression (disons qu'elle est au niveau 0). Il y a un seul opérande au niveau 1 (l'expression $x^a + y$), possédant lui-même deux arguments (x^a et y , au niveau 2). Enfin les noms x et a sont au niveau 3 de l'expression initiale.

Pour coder une expression en mémoire, Maple utilise une représentation arborescente. On peut en effet utiliser l'analogie avec un arbre. La fonction de tête est appelée la *racine* de l'arbre, et ses arguments en sont les *branches* qui peuvent elles-mêmes porter des *rameaux*, etc. On finit toujours par aboutir aux *feuilles* de l'arbre que sont les constantes ou les noms.

2.5.2. Les fonctions op et nops

Maple permet d'explorer en profondeur toute expression *expr* pour en extraire des éléments plus ou moins enfouis, ou pour remplacer un élément par un autre, etc.

Il faut pouvoir identifier l'objet de tête, en général une fonction ou un opérateur, qui est la racine de l'arbre, et ses opérandes immédiats qui en sont les branches. Il faut également savoir quel est le nombre de ces opérandes. On utilisera pour cela les fonctions **op** et **nops**.

- **nops(expr)** donne le nombre d'opérandes de l'expression *expr*.
- **op(expr)** donne la *séquence* de ces opérandes.
- **op(n, expr)** donne le *n*-ième de ces opérandes.
- **op(m..n, expr)** donne la *séquence* allant du *m*-ième au *n*-ième de ces opérandes.
- **op(0, expr)** donne la fonction ou l'opérateur de tête de l'expression.

2.5.3. Remplacements dans une expression

La fonction **op** permet donc de lire les composants d'une expression. Mais il est même possible de remplacer l'un d'eux par autre chose, avec les fonctions **subs** et **subsop**.

La fonction **subs** permet de remplacer une certaine sous-expression de *expr* par autre chose, et ceci à chaque fois qu'elle figure en tant qu'opérande (sans qu'on ait à savoir où).

- **subs(old = new, expr)**
remplace dans *expr* toute occurrence de l'expression *old* par l'expression *new*.

- $\text{subs}(old_1 = new_1, \dots, old_n = new_n, expr)$ remplace successivement old_1 par new_1 puis, dans le résultat, old_2 par new_2 , etc.
- $\text{subs}(\{old_1 = new_1, \dots, old_n = new_n\}, expr)$ ou $\text{subs}([old_1 = new_1, \dots, old_n = new_n], expr)$ réalisent des remplacements simultanés.

subsop permet de remplacer, dans une expression $expr$, un opérande de position donnée.

- $\text{subsop}(k=new, expr)$
remplace le k -ième opérande de $expr$ par l'expression new .
- $\text{subsop}(k_1=new_1, k_2=new_2, \dots, k_n=new_n, expr)$
remplace simultanément le k_1 -ième opérande par new_1 , le k_2 -ième par new_2 , etc.

2.5.4. Recherches dans une expression

La fonction has permet de savoir si une expression contient un objet particulier:

- $\text{has}(expr, obj)$ répond **true** ou **false** selon que obj figure ou non dans $expr$.
- $\text{has}(expr, \{obj_1, obj_2, \dots, obj_n\})$
répond **true** si l'un *au moins* des objets obj_1, \dots, obj_n figure dans $expr$.
- $\text{has}(expr, [obj_1, obj_2, \dots, obj_n])$ a la même signification.

Les fonctions lhs (*left hand side*) et rhs (*right hand side*) isolent les deux membres d'une relation, ou les extrémités d'un intervalle $a..b$ c'est-à-dire d'un objet de type *range*.

numer et denom extraient le numérateur et le dénominateur d'une expression algébrique, après mise au dénominateur commun, et simplification éventuelle.

La syntaxe est $\text{numer}(expr)$ et $\text{denom}(expr)$.

La fonction select permet de sélectionner les objets d'une expression (en fait une liste, un ensemble, une somme ou un produit) qui satisfont à un critère particulier. Ce critère est une fonction $test$ à une ou plusieurs variables, devant renvoyer un résultat booléen.

- $\text{select}(test, expr)$
renvoie une nouvelle expression (de même type que $expr$: ensemble, liste, somme ou produit) formée uniquement des objets obj de $expr$ pour lesquels $test(obj)$ vaut **true**.
- $\text{select}(test, expr, arg_2, \dots, arg_n)$
joue le même rôle, mais dans le cas où $test$ est une fonction de n variables, et conserve les objets obj de $expr$ pour lesquels $test(obj, arg_2, \dots, arg_n)$ vaut **true**.

La fonction indets recherche les objets indéterminés (symboliques) d'une expression:

- $\text{indets}(expr)$ renvoie l'ensemble des objets indéterminés de $expr$.
Seuls les opérateurs $+, -, *, /$ sont traversés dans cette recherche.
- $\text{indets}(expr, typename)$ renvoie les sous-objets de $expr$ qui sont de type *typename*.

2.5.5. Mapper une fonction dans une expression

L'instruction **map** permet d'appliquer une même fonction *func* (d'une variable) à tous les opérandes d'une expression. Plus précisément, **map**(*func*, *expr*) remplace dans *expr* tous les opérandes *obj* par *func*(*obj*).

Il est même possible de mapper une fonction de *n* variables, mais il faut alors préciser la valeur des *n* − 1 arguments supplémentaires : ainsi **map**(*func*, *expr*, *arg*₂, ..., *arg*_{*n*}) remplace dans *expr* tous les opérandes *obj* par *func*(*obj*, *arg*₂, ..., *arg*_{*n*}).

Il est également possible d'appliquer une même fonction binaire aux éléments de deux listes ou de deux vecteurs, avec la fonction **zip**.

Les opérateurs arithmétiques sont automatiquement *mappés* sur les égalités ou les inégalités : pour ces dernières, le produit peut réserver des surprises!

Il existe d'autres possibilités de substitution dans Maple. On pourra consulter l'aide en ligne sur les fonctions **asubs** (qui doit être chargée par **readlib**) et **powsubs** (qui fait partie du package **student**) et chercher en quoi elles diffèrent de la fonction **subs**.

2.5.6. Reconnaissance de motif

La fonction **match** permet de savoir si une expression obéit à un certain motif (*pattern*) :

match(*expr*=*motif*, *var*, '*eqs*') répond **true** ou **false** selon que *expr* obéit ou non au *motif*.

Le nom *var* désigne la variable principale, tant dans *expr* que dans *motif*.

Si *motif* est reconnue, *eqs* est affecté d'une liste d'égalités {*old*₁ = *new*₁, ..., *old*_{*n*} = *new*_{*n*}}.

Cette liste est telle que l'instruction **subs**(*eqns*, *pattern*) redonnerait l'expression *expr*.

Le troisième argument doit être un nom. Il est donc préférable de l'entourer d'apostrophes (en tout cas c'est obligatoire si ce nom a déjà été affecté).

2.6. EXPRESSIONS : EXEMPLES

2.6.1. Les fonctions op et nops

On considère ici l'expression $f = \ln(x)^{a^c}$.

L'opérateur racine est l'exponentiation, qui possède les deux opérandes $\ln x$ et a^c .

`op(1,op(2,f))` donne le premier opérande du deuxième opérande de f .

Il s'agit donc du premier opérande de a^c , c'est-à-dire a .

```
> restart:  f:=ln(x)^(a^c):
           op(0,f), nops(f); op(f); op(1,f); op(1,op(2,f));
```

$$\begin{array}{c} ^, 2 \\ \ln(x), a^c \\ \ln(x) \\ a \end{array}$$

On forme ici l'expression $func(a, b, c * d, e, x, \frac{y}{z}, t, u)$.

On extrait ensuite la séquence des opérandes de position comprise entre 2 et 6 :

```
> restart:  expr:=func(a,b,c*d,e,x,y/z,t,u):  op(2..6,expr);
```

$$b, c d, e, x, \frac{y}{z}$$

On prendra garde au fait que pour les opérateurs commutatifs et associatifs (comme `+`, `*`, `min`, `max`, `union`, etc.), l'ordre des arguments peut varier d'une session à l'autre.

L'ordre dans lequel ils sont réarrangés par Maple est en principe celui dans lequel il sont apparus pour la première fois dans la session en cours (ou depuis le dernier `restart`).

En particulier, s'ils ont déjà été utilisés, ils ne figureront pas forcément dans l'ordre où ils sont saisis lors de la création de l'expression par l'utilisateur. On le voit bien ci-dessous avec la première définition de `g`.

```
> restart:  f:=d+b+a+c; g:=a+b+c+d; op(g);
restart:  g:=a+b+c+d; op(g);
```

$$\begin{array}{c} f := d + b + a + c \\ g := d + b + a + c \\ d, b, a, c \\ g := a + b + c + d \\ a, b, c, d \end{array}$$

Autre risque d'erreur : la liste des opérandes d'un ensemble doit être considérée après évaluation de celui-ci (et donc suppression des éventuels *doublons*).

De plus, l'ordre des éléments dans un même ensemble peut varier d'une session à l'autre.

Sur l'exemple suivant, on voit que la *liste* $[a, b, c, a, c, a]$ et l'*ensemble* $\{a, b, c, a, c, a\}$, bien que créés avec les mêmes éléments, n'ont pas les mêmes opérandes après évaluation : les doublons ont en effet été éliminés de l'ensemble.

```
> restart:  [a,b,c,a,c,a]:  op(%); {a,b,c,a,c,a}:  op(%);
```

$$a, b, c, a, c, a$$

$$a, b, c$$

On notera que les quotients formels a/b sont en fait mémorisés sous la forme $a * b^{-1}$.

L'opérateur racine est donc $*$ et non $/$. On le voit bien ici avec $expr = \frac{a + b * c}{x * y + z}$:

```
> restart:  expr:=(a+b*c)/(x*y+z);
          op(0,expr), nops(expr); op(expr);
```

$$expr := \frac{a + b c}{x y + z}$$

$$*, 2$$

$$a + b c, \frac{1}{x y + z}$$

Le deuxième opérande de cette expression est $(x * y + z)^{-1}$.

Son opérateur racine est l'exponentiation, et ses opérandes sont $x * y + z$ et -1 .

```
> op(0,op(2,expr)); op(op(2,expr));
```

$$^$$

$$x y + z, -1$$

Si $expr = f(arg)$ où f est une fonction d'une variable, $op(expr)$ équivaut à $op(1, expr)$:

```
> expr:=cos(tan(sin(Pi^2))); op(1,op(op(op(expr))));
```

$$expr := \cos(\tan(\sin(\pi^2)))$$

$$\pi$$

2.6.2. Remplacements par subs

Dans l'expression $a + \cos(a + \sin(1 + a))$, on remplace partout a par $b + c$:

```
> restart: subs(a=b+c,a+cos(a+sin(1+a)));
```

$$b + c + \cos(b + c + \sin(1 + b + c))$$

Attention! Substitution ne signifie pas assignation. On le voit avec l'exemple suivant : le fait de remplacer x par 10 ne veut pas dire qu'on a placé 10 dans la variable x .

```
> restart: subs(x=10,1+x), x;
```

$$11, x$$

Autre problème potentiel : on doit faire attention aux assignations déjà effectuées.

L'expression à traiter par **subs** est en effet évaluée avant les substitutions.

Dans l'exemple ci-dessous, on place d'abord 5 dans la variable x . On tente ensuite de remplacer x par 10 dans l'expression $1 + x$. Mais celle-ci est d'abord évaluée en $1 + 5 = 6$, et aucune substitution de x n'est plus possible.

```
> x:=5; subs(x=10,1+x), x;
```

$$x := 5$$

$$6, 5$$

Après les remplacements par **subs**, la nouvelle expression est *simplifiée* mais pas *évaluée*. On voit par exemple que remplacer a par 5 dans $\min(a, 3)$ produit l'expression $\min(5, 3)$ mais que cette dernière n'est pas automatiquement évaluée :

```
> [subs(a=5,1+a^2), subs(a=5,min(a,3)), subs(a=Pi/3,sin(a))];  
eval(%);
```

$$[26, \min(3, 5), \sin(\frac{1}{3}\pi)]$$

$$[26, 3, \frac{1}{2}\sqrt{3}]$$

Puisque substitution ne signifie pas assignation, l'exemple suivant ne crée pas de récursivité :

```
> subs(x=x+1,x^3+2*x^2+x-5); expand(%);
```

$$(x + 1)^3 + 2(x + 1)^2 + x - 4$$

$$x^3 + 5x^2 + 8x - 1$$

Les substitutions suivantes ne fonctionnent pas car aucune des expressions $x + y$, $x + z$, $y + z$ n'est un *opérande* de l'expression $x + y + z$ (ni un opérande d'un opérande, etc.).

En fait l'opérateur racine de la somme $x + y + z$ est $+$ et ses opérandes sont x , y et z .

> restart:

```
subs(x+y=a,x+y+z), subs(x+z=a,x+y+z), subs(y+z=a,x+y+z);
```

$$x + y + z, x + y + z, x + y + z$$

Ici $x + y$ figure comme *opérande* (de manière plus ou moins interne) à deux reprises:

> restart: subs(x+y=a,sin(x+y)+x+y+z/(x+y));

$$\sin(a) + x + y + \frac{z}{a}$$

On se méfiera des simplifications automatiques, comme le montre le deuxième exemple ci-dessous, dans lequel l'expression $2 * (x + y) + z$ est d'abord *simplifiée* en $2 * x + 2 * y + z$ (et alors $x + y$ n'est plus un *opérande*) :

> restart: subs(x+y=a,2^(x+y)+z), subs(x+y=a,2*(x+y)+z);

$$2^a + z, 2x + 2y + z$$

Voici quelques exemples de remplacements successifs.

Dans l'expression $f(x, y)$, si on remplace x par y , puis y par x , on obtient $f(x, x)$.

Dans $x + z + \sin xy$:

- Si on remplace x par y , on obtient $y + z + \sin y^2$.
- Si on remplace ensuite z par x on obtient $y + x + \sin y^2$.
- Si on remplace ensuite x par a on obtient $y + a + \sin y^2$.

> restart:

```
subs(x=y,y=x,func(x,y)), subs(x=y,z=x,x=a,x+z+sin(x*y));
```

$$\text{func}(x, x), y + a + \sin(y^2)$$

Dans $f(x, y)$:

- Si on remplace x par $x + y$, on obtient $f(x + y, y)$.
- Si on remplace ensuite y par $y - x$ on obtient $f(y, y - x)$.
- Si on remplace ensuite x par $y - x$ on obtient $f(y, x)$.

> restart: subs(x=x+y,y=y-x,x=y-x,func(x,y));

$$\text{func}(y, x)$$

Voici maintenant un exemple de remplacements simultanés.
On voit comment effectuer l'échange de x et de y dans $f(x, y)$.

```
> restart:
subs([x=y,y=x],func(x,y)); subs({x=y,y=x},func(x,y));

func(y, x)

func(y, x)
```

2.6.3. Remplacements par subsop

Dans cet exemple, on remplace le premier argument de $f(a, b, c, d)$ par 2.

Dans la même expression, on remplace le deuxième argument par x et le troisième par y .

```
> restart: func(a,b,c,d); subsop(1=2,%), subsop(2=x,3=y,%);

func(a, b, c, d)

func(2, b, c, d), func(a, x, y, d)
```

Comme avec `subs`, le résultat peut subir des simplifications mais il n'est pas évalué.
Ici on remplace le premier argument z par 2. Le résultat 2^3 est simplifié.

```
> restart: subsop(1=2,z^3);
```

8

Ici on remplace le premier argument par b , ce qui conduit à $\min(b, b, c, d)$.
Il faut utiliser `eval` pour que cette expression soit réduite en $\min(b, c, d)$.

```
> restart: min(a,b,c,d); subsop(1=b,%), eval(subsop(1=b,%));

min(a, b, c, d)

min(b, b, c, d), min(b, c, d)
```

On doit se méfier de l'ordre des arguments pour les opérateurs commutatifs et associatifs.

Cet ordre correspond en principe à l'ordre d'entrée en scène dans la session en cours.

Ici x se situe avant y .

On a beau saisir $g := y + x$, le contenu de g est mémorisé sous la forme $x + y$.

L'instruction `subsop(1=a,g)` remplace le premier argument de g par a , mais ce premier argument est x et non y comme on aurait pu le croire.

```
> restart: f:=x+y: g:=y+x: subsop(1=a,g);

a + y
```

La position des opérandes n'est pas toujours évidente.

Ici on peut penser que le deuxième argument est z . En fait c'est $\frac{1}{z}$.

On en voit la confirmation avec l'instruction `op`.

```
> restart:  f:=(x+y)/z:  subsop(2=a,f); op(f);
```

$$(x + y) a$$

$$x + y, \frac{1}{z}$$

Avec `subsop(0 = new, expr)` on peut même changer la tête d'une expression :

```
> restart:  g:=min(a,b,c):  subsop(0=max,g);
```

$$\max(a, b, c)$$

Voici deux exemples de remplacements simultanés avec `subsop`.

On remplace le premier argument, c'est-à-dire a^b , par x .

On remplace en même temps le deuxième argument, c , par y .

```
> restart:
subsop(1=x,2=y,(a^b)^c); subsop(1=x,3=y,max(a,b,c,d));
```

$$x^y$$

$$\max(x, a, y, d)$$

2.6.4. Recherches dans une expression

Rappelons que $x + y$ n'est pas une sous-expression de $x + y + z$ (celles-ci sont x , y et z). Cela explique le résultat (*false*) de l'instruction `has(x+y+z,x+y)`.

```
> restart:  has(x+y+z,x), has(x+y+z,x+y), has(x+z+sin(x+y),x+y);
```

$$true, false, true$$

x et x^3 sont des sous-expressions de $x^3 + 4$, mais pas x^2 .

```
> restart:  f:=x^3+4; has(f,x), has(f,x^2), has(f,x^3);
```

$$f := x^3 + 4$$

$$true, false, true$$

Attention à l'évaluation qui précède l'exécution de **has** !

Ici f est saisi en $(x^2)^2$ mais évalué en x^4 . Donc x^2 n'est pas une sous-expression de f .

```
> restart:  f:=(x^2)^2:  has(f,x^2);
```

false

Voici un exemple où on teste la présence d'au moins une sous-expression.

f ne contient pas $x + y$ (dans $1 + x + y$ les seules sous-expressions sont 1, x , et y .)

En revanche f contient la sous-expression $x - y$ (comme unique argument de **exp**).

```
> restart:
```

```
  f:=(x*(1+x+y)+2)/(x+exp(x-y)):  has(f,x+y),has(f,[x+y,x-y]);
```

false, true

Voici comment extraire les membres gauche ou droit d'une égalité ou d'une inégalité :

```
> eq:=a=b:  eq, lhs(eq), rhs(eq);
```

```
  ineq:=x+y<t*u:  ineq, lhs(ineq), rhs(ineq);
```

$a = b, a, b$

$x + y < t u, x + y, t u$

Attention, les inégalités $>$ et $>=$ sont automatiquement converties en $<$ et $<=$:

```
> ineq:=x+y>t*u:  ineq, lhs(ineq), rhs(ineq);
```

$t u < x + y, t u, x + y$

Les opérateurs de relation $<$, $>$, etc. ont la même priorité.

Celle-ci est plus faible que celle des opérateurs arithmétiques.

En revanche elle est plus forte que celle des opérateurs logiques **not**, **and**, **or**, et que celle de l'opérateur *virgule* (séparateur dans une séquence).

Ici eq est la séquence formée d'abord de $x = z$ puis de c , ce qui explique l'échec de **lhs** et **rhs**.

```
> restart:  eq:=x=z,c:  lhs(eq); rhs(eq);
```

```
Error, (in lhs) invalid arguments
```

```
Error, (in rhs) invalid arguments
```

Ce nouvel exemple corrige l'erreur du précédent :

```
> restart:  eq:=x=(z,c):  lhs(eq); rhs(eq);
```

x

z, c

Les instructions `lhs` et `rhs` peuvent servir à extraire les extrémités d'un intervalle :

```
> a:=5..8; lhs(a), rhs(a), lhs(a)..rhs(a)+1;
```

$$a := 5..8$$

$$5, 8, 5..9$$

Voici comment extraire le numérateur et le dénominateur d'une expression *expr*.

Là encore, il faut compter avec l'évaluation initiale de *expr*.

```
> denom(3/4), numer(15/11), denom(14/10),
  denom(14/7), numer(1+1/2+1/3);
```

$$4, 15, 5, 1, 11$$

```
> restart: f:=1+1/x+1/(y+1); numer(f), denom(f);
```

$$f := 1 + \frac{1}{x} + \frac{1}{y+1}$$

$$x y + 2 x + y + 1, x (y + 1)$$

Voici comment rechercher les sous-expressions *indéterminées* dans une expression donnée, pour les opérandes arithmétiques `+`, `-`, `*`, et `/`.

La syntaxe `indets(f,name)` donne les *noms* indéterminés dans l'expression.

```
> restart: f:=x+y*cos(t)+exp(z+x); indets(f), indets(f,name);
```

$$f := x + y \cos(t) + e^{(z+x)}$$

$$\{\cos(t), e^{(z+x)}, x, y, t, z\}, \{x, y, t, z\}$$

2.6.5. Sélections dans une expression

Dans la liste `[1, 5, 3, 1, 2, 1, 4]` on sélectionne les éléments supérieurs à 2 :

```
> select(x->x>2, [1,5,3,1,2,1,4]);
```

$$[5, 3, 4]$$

On extrait ici les entiers premiers compris entre 100 et 150 :

```
> select(isprime, {100..150});
```

$$\{137, 139, 149, 127, 131, 103, 107, 109, 113, 101\}$$

On sélectionne maintenant les entiers impairs dans la liste `[17, 24, 57, 98, 113]` :

```
> select(x->x mod 2=1, [17,24,57,98,113]);
```

$$[17, 57, 113]$$

Voici une autre manière de traiter le problème précédent :

```
> select((x,y)->x mod y=1,[17,24,57,98,113],2);
```

$$[17, 57, 113]$$

Dans l'expression $1 + x + y + x^2 + xz + \frac{y}{x}$, on ne garde que les sous-expressions contenant x :

```
> restart: select(has,1+x+y+x^2+x*z+y/x,x);
```

$$x + x^2 + xz + \frac{y}{x}$$

2.6.6. Mapper une fonction dans une expression

Quand on mappe par exemple la fonction $x \rightarrow x + 1$ sur tous les opérandes d'une expression, on ne doit pas confondre le x utilisé dans la définition de cette fonction (et qui est une variable muette) et les x présents éventuellement dans l'expression.

```
> restart:
map(x->x+1,x^y), map(x->x+1,x+y+z), map(x->1/x,x+y+z);
```

$$(x + 1)^{(y+1)}, x + 3 + y + z, \frac{1}{x} + \frac{1}{y} + \frac{1}{z}$$

On voit ici comment les fonctions `sqrt` ou `sin` sont appliquées aux arguments a, b, c .

```
> restart: map(sqrt,f(a,b,c)), map(sin,a+b+c);
```

$$f(\sqrt{a}, \sqrt{b}, \sqrt{c}), \sin(a) + \sin(b) + \sin(c)$$

Dans cet exemple, la fonction à mapper exige deux arguments.

Le premier est extrait successivement de la liste $[1, 5, 2, 7, 3, 2, 1, 6]$.

Le deuxième est toujours pris égal à 3.

On remplace donc chaque élément x de la liste par $\min(x, 3)$.

```
> map((x,y)->min(x,y),[1,5,2,7,3,2,1,6],3);
```

$$[1, 3, 2, 3, 3, 2, 1, 3]$$

Voici un exemple un peu plus élaboré.

Chaque élément de la liste est :

- Conservé s'il est compris entre 3 et 7.
- Remplacé par 3 s'il est inférieur à 3.
- Remplacé par 7 s'il est supérieur à 7.

```
> map((x,y,z)->max(y,min(x,z)),[1,5,10,4,2,8,6,5,7,3],3,7);
```

$$[3, 5, 7, 4, 3, 7, 6, 5, 7, 3]$$

On voit comment les opérateurs $-$ et $+$ sont automatiquement mappés sur les inégalités.

Si on part d'inégalités vraies, celles auxquelles on aboutit sont vraies aussi.

```
> restart:  -(a<b), (y<z)+(a<b), (y<z)-(a<b), (x<y)+(a>b);
```

$$-b < -a, y + a < z + b, y - b < z - a, x + b < y + a$$

En revanche (voir le premier exemple), on prendra garde au fait que le produit est lui aussi automatiquement mappé sur les inégalités.

```
> restart:
```

```
x*(y<z), 0*(y<z), 2*(x=y)+3*(a=b), map(x->1/x,a=b);
```

$$x y < x z, 0 = 0, 2 x + 3 a = 2 y + 3 b, \frac{1}{a} = \frac{1}{b}$$

On ne peut pas directement ajouter un terme au deux membres d'une égalité.

```
> restart:  x+(y=z);
```

```
Error, (in simpl/relosum) invalid terms in sum
```

Voici comment régler le problème précédent :

```
> restart:  (x=x)+(y=z);
```

$$x + y = x + z$$

2.6.7. Reconnaissance de motifs

On demande à Maple si l'expression $5x^2 + 7x + 3$ est du type $ax^2 + bx + c$.

La réponse est oui, avec $c = 3$, $a = 5$ et $b = 7$.

```
> restart:  match(5*x^2+7*x+3=a*x^2+b*x+c,x,'eqs'); eqs;
```

true

$$\{c = 3, a = 5, b = 7\}$$

Ici on vérifie que $y \exp \frac{1}{y}$ est du type $(ay + b) \exp \frac{c}{y}$, avec $b = 0$, $a = 1$ et $c = 1$.

```
> restart:  match(y*exp(1/y)=(a*y+b)*exp(c/y),y,'eqs'); eqs;
```

true

$$\{b = 0, a = 1, c = 1\}$$

2.7. SÉQUENCES, LISTES, ENSEMBLES

2.7.1. Séquences

Une *séquence* est une succession d'objets séparés par des virgules. Cette séquence peut être vide : elle est alors désignée par le nom `NULL`.

La principale instruction permettant de créer une séquence est `seq` :

- `seq(f(k), k = m..n)` construit la séquence des $f(k)$ quand k prend toutes les valeurs de m à n inclus : m , puis $m + 1$, etc. ; m et n peuvent ne pas être entiers.
- `seq(f(k), k = expr)` joue le même rôle, k prenant comme valeurs successives les différents opérandes de l'expression *expr*.

En général *expr* est un *ensemble* ou une *liste*.

La virgule qui sépare les éléments d'une séquence doit être considérée comme un opérateur. Sa précedence très faible (toutes les opérations sont donc évaluées avant la prise en compte de la séparation) mais tout de même plus élevée que celle de l'opérateur d'assignation `:=`.

Pour créer des séquences, on peut également utiliser l'opérateur `$`.

- `expr $ n` crée la séquence formée de n exemplaires de *expr*.
- `f(k) $ k = m..n` est équivalent à `seq(f(k), k = m..n)`
- `$ m..n` crée la séquence $m, m + 1, m + 2, \dots$, jusqu'à la dernière valeur $\leq n$.

Dans les instructions de création de listes par `$`, il est recommandé de placer le nom de l'indice (k dans l'exemple ci-dessus), ainsi que l'expression $f(k)$, entre apostrophes. C'est en fait obligatoire si le nom de l'indice est déjà assigné.

2.7.2. Listes

Une *liste* est une séquence éventuellement vide délimitée par les caractères `[` et `]`.

Les éléments d'une liste peuvent aussi être des listes.

Une liste est une structure ordonnée (les éléments restent dans le même ordre, tant que l'utilisateur ne décide pas de le modifier) et acceptant les répétitions (tous les éléments peuvent même être égaux).

L'instruction `sort` permet de réordonner les éléments d'une liste L :

- `sort(L)` trie la liste L dans l'ordre croissant pour une liste de nombres, dans l'ordre lexicographique croissant pour une liste de chaînes, et sinon par ordre d'adresse croissante en mémoire.
- `sort(L, test)` trie L suivant le résultat donné par *test*, fonction de deux variables renvoyant une valeur booléenne.

Si a et b sont deux éléments de L , a sera placé avant b si `test(a, b)` renvoie `true`.

Il y a des cas particuliers pour cette fonction *test* :

- Si *test* vaut '`<`' ou `numeric`, le tri s'effectue suivant les valeurs numériques croissantes.
- Si *test* vaut `lexorder` ou `string` c'est l'ordre lexicographique croissant.

L'instruction `convert(L,multiset)` permet de savoir combien de fois chaque élément figure dans une liste *L* : le résultat est la liste des `[obj,num]`, où *obj* est un élément de *L* et où *num* est le nombre de fois où il y figure.

2.7.3. Ensembles

Un *ensemble* (*set* en anglais) est une séquence (éventuellement vide) délimitée par { et }.

Les éléments d'un ensemble peuvent aussi être des ensembles.

Un ensemble est une structure non ordonnée : les éléments d'un ensemble apparaissent dans un ordre dépendant de la session en cours. L'utilisateur ne peut pas, d'une session à l'autre, être certain que l'ordre ne sera pas changé.

Quand un ensemble est évalué, les éléments en double sont supprimés : on est donc très près de la notion mathématique d'ensemble.

Les opérateurs `union`, `intersect` et `minus` permettent d'effectuer des opérations entre ensembles. Si s_1, s_2, \dots, s_n sont des ensembles :

- $s_1 \text{ union } s_2 \dots \text{ union } s_n$ donne la réunion de s_1, s_2, \dots, s_n .
On peut aussi utiliser la syntaxe '`union`'(s_1, s_2, \dots, s_n).
- $s_1 \text{ intersect } s_2 \dots \text{ intersect } s_n$ donne leur intersection.
Une syntaxe équivalente est '`intersect`'(s_1, s_2, \dots, s_n).
- $s_1 \text{ minus } s_2$ donne l'ensemble des éléments qui sont dans s_1 et pas dans s_2 .
On peut encore écrire '`minus`'(s_1, s_2).

Les opérateurs `union` et `minus` ont la même priorité, inférieure à celle de `intersect`.

2.7.4. Passer d'une structure à l'autre

Soit *S* une séquence. On forme une liste en écrivant `[S]` et un ensemble en écrivant `{S}`.

Si *X* est une liste ou un ensemble, `op(X)` donne la séquence de ses éléments.

On transformera donc la liste *L* en un ensemble par `{op(L)}`.

De même on convertira l'ensemble *E* en une liste en écrivant `[op(E)]`.

On peut aussi utiliser `convert` en écrivant `convert(L,set)` ou `convert(E,list)`.

2.7.5. Rechercher des éléments

L'instruction `nops` calcule le nombre d'éléments d'une liste *L* ou d'un ensemble *E*.

Cette syntaxe ne convient pas pour les séquences : en effet si *S* est par exemple la séquence *a, b, c* alors `nops(S)` s'écrit `nops(a,b,c)` et est interprété par Maple comme un appel de la fonction `nops` avec trois arguments.

Pour dénombrer une séquence S , on écrira donc `nops([S])`. Cela revient à passer par une liste. On ne doit pas passer par un ensemble si on ne veut pas risquer de perdre le décompte des éléments répétés dans S .

Si X est une séquence (ou un ensemble ou une liste), alors :

- $X[n]$ donne le n -ième élément de X .
- $X[m..n]$ donne la séquence (ou l'ensemble ou la liste) du m -ième au n -ième élément de X .

Si X est un ensemble ou une liste mais pas une séquence, on peut aussi écrire `op(n, X)` ou `op(m..n, X)` pour extraire un ou plusieurs éléments consécutifs. On peut tester l'appartenance

d'un élément à un ensemble ou une liste (pas une séquence) :

- `member(expr, X)` donne `true` si la valeur de *expr* figure dans X , et `false` sinon.
- `member(expr, X, 'name')` joue le même rôle, mais en plus, si le résultat est `true`, alors la position où a été trouvé l'élément est renvoyée dans la variable *name*.

On notera les apostrophes autour de *name*, toujours recommandées, mais nécessaires si ce nom est déjà assigné.

2.7.6. Ajouter, remplacer ou enlever des éléments

Soit L une liste :

- On peut ajouter un élément x à la fin de L en écrivant `[op(L), x]`.
- Pour remplacer le k -ième élément par *new*, on écrira `subsop(k = new, L)`.
- Pour éliminer le k -ième élément : `subsop(k = NULL, L)`.
- Pour *insérer* un nouvel élément x en position k : `subsop(k = (x, L[k]), L)`.

Soient L_1 et L_2 deux listes :

- Pour les *concaténer*, on écrira `[op(L1), op(L2)]`.
- Pour *insérer* L_2 à partir de la position k de L_1 , on écrira :
`subsop(k = (op(L2), L1[k]), L1)`.

Soit E un ensemble :

- On peut lui ajouter un élément x par : E `union` $\{x\}$ ou $\{op(E), x\}$.
- On peut retirer l'élément x de E en écrivant E `minus` $\{x\}$.

2.7.7. Mapper des opérations sur des ensembles ou des listes

- On sait que `map` permet d'appliquer une même fonction f d'une variable aux éléments d'un ensemble ou d'une liste X .

La syntaxe est `map(f, X)`, et le résultat est l'ensemble (resp. la liste) des images par f des différents éléments de X .

La syntaxe `map($f, X, arg_2, \dots, arg_n$)` applique à tous les éléments obj de X la fonction de n variables f , le résultat étant l'ensemble ou la liste des images $f(obj, arg_2, \dots, arg_n)$.

- L'instruction `zip` permet d'appliquer une même fonction f de *deux* variables à deux listes $L_1 = [a, b, c, \dots]$ et $L_2 = [x, y, z, \dots]$.

La syntaxe est `zip(f, L_1, L_2)`, et le résultat est la liste $[f(a, x), f(b, y), f(c, z), \dots]$.

Si L_1 et L_2 n'ont pas même longueur, `zip` tronque la liste la plus longue, à moins qu'un quatrième argument ne soit présent: avec la syntaxe `zip(f, L_1, L_2, def)`, l'objet def est utilisé comme argument par défaut pour compléter celle des deux listes qui est la plus courte.

2.8. SÉQUENCES, LISTES, ENSEMBLES : EXEMPLES

2.8.1. Séquences

Voici un exemple de *concaténation* de séquences :

```
> s:=1,2,4,8:  t:=s,16:  u:=t,s;
```

$$u := 1, 2, 4, 8, 16, 1, 2, 4, 8$$

On forme la séquence de tous les k , à partir de $k = 1.7$, de 1 en 1, sans dépasser 8.3 :

```
> seq(k,k=1.7..8.3);
```

$$1.7, 2.7, 3.7, 4.7, 5.7, 6.7, 7.7$$

Voici la séquence des $k(k+1)$, de $k = 1$ à $k = 5$:

```
> restart:  seq(k*(k+1),k=1..5);
```

$$2, 6, 12, 20, 30$$

On reprend le même exemple, mais en donnant au préalable la valeur 10 à la variable k .
On voit que le fonctionnement de `seq` n'en est pas pour autant perturbé.

```
> k:=10:  seq(k*(k+1),k=1..5);
```

$$2, 6, 12, 20, 30$$

Après un `restart`, on forme encore la même séquence, cette fois avec l'opérateur `$` :

```
> restart:  k*(k+1) $ k=1..5;
```

$$2, 6, 12, 20, 30$$

Nouvel essai, après avoir donné une valeur à la variable k .

Maple répond ici par un message d'erreur.

En effet, il évalue l'expression en "`110 $ 10=1..5`" avant d'évaluer l'opérateur `$`.

Contrairement à l'instruction `seq`, l'opérateur `$` ne *protège* donc pas l'indice de sommation.

Il aurait fallu écrire '`k*(k+1)`' \$ '`k`'=1..5.

Si on écrit `k*(k+1) $ 'k'=1..5`, on obtient la séquence 110, 110, 110, 110, 110.

```
> k:=10:  k*(k+1) $ k=1..5;
```

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

L'opérateur \$ est plus indiqué pour créer rapidement ce genre de séquence :

```
> $3..7; 7$5;
```

3, 4, 5, 6, 7

7, 7, 7, 7, 7

Voici comment calculer le nombre d'éléments dans une séquence :

```
> restart: s:=a,e,d,c,e,c,a,a,b,c; nops([s]);
```

$s := a, e, d, c, e, c, a, a, b, c$

10

Voici maintenant comment calculer le nombre d'éléments *différents* dans une séquence s .

On convertit d'abord s en un ensemble pour en éliminer tous les doublons.

```
> restart: s:=a,e,d,c,e,c,a,a,b,c; nops({s});
```

$s := a, e, d, c, e, c, a, a, b, c$

5

2.8.2. Listes

On place une liste dans la variable L. On voit que les éléments restent dans l'ordre défini par l'utilisateur et qu'il peut y avoir des doublons. L'instruction `convert(L,multiset)` permet de savoir combien de fois chaque élément figure dans cette liste.

```
> restart: L:=[a,b,c,d,c,a,c]: nops(L); op(L); convert(L,multiset);
```

7

a, b, c, d, c, a, c

$[[c, 3], [d, 1], [a, 2], [b, 1]]$

L'instruction `sort` trie les listes de nombres par ordre croissant.

Voici comment les trier par ordre décroissant :

```
> L:=[1,9,3,4,10,5]; sort(L), sort(L,(a,b)->evalb(a>b));
```

$L := [1, 9, 3, 4, 10, 5]$

$[1, 3, 4, 5, 9, 10], [10, 9, 5, 4, 3, 1]$

Les listes de noms sont triées par ordre lexicographique croissant :

```
> restart:  L1:=[a,g,c,f,b,h]; L2:=sort(L1);
```

$$L1 := [a, g, c, f, b, h]$$

$$L2 := [a, b, c, f, g, h]$$

Pour une liste mêlant des nombres et des noms, le tri peut être surprenant :

```
> restart:  e:=[x,10,y,a,30,z,20]; sort(e);
```

$$e := [x, 10, y, a, 30, z, 20]$$

$$[x, 30, y, a, z, 20, 10]$$

2.8.3. Ensembles

Lors de la première évaluation d'un ensemble, les doublons sont éliminés :

```
> restart:  e:={a,b,c,d,c,a}:  nops(e); op(e);
```

$$4$$

$$a, b, c, d$$

L'ordre dans lequel les éléments sont stockés puis affichés dépend de l'adresse en mémoire.

On ne peut donc garantir que cet ordre sera conservé d'une session à l'autre.

Le point positif est que deux ensembles formés des mêmes éléments, mais saisis dans un ordre différent par l'utilisateur, sont mémorisés et affichés de la même manière.

```
> restart:  e1:={d,a,c,b}:  e2:={a,b,c,d}:  e1, e2, evalb(e1=e2);
```

$$\{c, a, b, d\}, \{c, a, b, d\}, true$$

Voici quelques exemples d'opérations sur des ensembles :

```
> restart:  e1:={a,b,c,d}:  e2:={b,c,e}:  e3:={a,c,f}:
```

```
> e1 intersect e2, e2 union e3, e1 minus e2; 'union'(e1,e2,e3);
```

$$\{b, c\}, \{a, b, c, e, f\}, \{a, d\}$$

$$\{a, b, c, d, e, f\}$$

Voici une liste et un ensemble formé des mêmes éléments.

On le voit, l'ordre des éléments de l'ensemble peut être quelconque.

```
> [1..20];
{1..20};
```

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]$$

$$\{1, 2, 9, 10, 11, 12, 13, 14, 3, 4, 5, 6, 7, 8, 20, 15, 16, 17, 18, 19\}$$

2.8.4. Passer d'une structure à l'autre

Ici on convertit une liste ou un ensemble en la séquence de ses éléments :

```
> restart:
op([d,b,a,c]);
op({a,b,c,d});
```

$$d, b, a, c$$

$$d, b, a, c$$

Voici deux manières équivalentes de transformer une liste en un ensemble :

```
> restart:
[a,c,b,d];
convert(%,set),{op(%)};
```

$$[a, c, b, d]$$

$$\{a, c, b, d\}, \{a, c, b, d\}$$

Ici, on voit comment éliminer tous les doublons dans une liste L .

Il suffit de convertir cette liste en un ensemble avant de revenir à une liste :

```
> restart:
L:=[f,e,a,c,a,e,b,a,f];
[op({op(L)})];
```

$$L := [f, e, a, c, a, e, b, a, f]$$

$$[f, e, a, c, b]$$

2.8.5. Rechercher des éléments

On forme ici une séquence s , puis on la convertit en une liste L et en un ensemble e .

On extrait ensuite le quatrième élément de s , de L , et de e .

```
> restart:  s:=a,x,b,y,c,z; L:=[s]; e:={s}; s[4], L[4], e[4];
```

$$s := a, x, b, y, c, z$$

$$L := [a, x, b, y, c, z]$$

$$e := \{x, y, a, b, c, z\}$$

$$y, y, b$$

A la suite de l'instruction précédente, on modifie s , puis on la convertit à nouveau en L et e .

On extrait alors la séquence (resp. la liste, l'ensemble) des éléments situés entre la deuxième et la quatrième position :

```
> s:=a,b,c,x,y,z; L:=[s]; e:={s}; s[2..4]; L[2..4]; e[2..4];
```

$$s := a, b, c, x, y, z$$

$$L := [a, b, c, x, y, z]$$

$$e := \{x, y, a, b, c, z\}$$

$$b, c, x$$

$$[b, c, x]$$

$$\{y, a, b\}$$

Voici comment on peut encore extraire un élément, ou une séquence d'éléments consécutifs.

```
> restart:  L:=[a,x,b,y,c,z]:  op(5,L); op(4..6,L);
```

$$c$$

$$y, c, z$$

On teste ici la présence d'un élément dans une liste.

On voit comment l'argument optionnel permet de savoir où il se situe.

```
> restart:  L:=[a,x,b,y,c,z]:  member(t,L); member(c,L,'pos'), pos;
```

$$false$$

$$true, 5$$

2.8.6. Ajouter, remplacer ou enlever des éléments

On concatène deux listes L_1 et L_2 :

```
> restart:  L1:=[a,b,c,d]:  L2:=[x,y,z]:  [op(L1),op(L2)];
```

$$[a, b, c, d, x, y, z]$$

On voit comment ajouter un élément x en fin ou en début de la liste L .

On voit aussi comment placer cet élément en une position donnée :

```
> x:='x':  L:=[1,2,3,4]:  [op(L),x], [x,op(L)], subsop(3=x,L);
```

$$[1, 2, 3, 4, x], [x, 1, 2, 3, 4], [1, 2, x, 4]$$

Ici, on supprime le deuxième élément de la liste L .

On voit aussi comment insérer un élément en troisième position :

```
> restart:  L:=[a,b,c,d]:  subsop(2=NULL,L), subsop(2=(b,z),L);
```

$$[a, c, d], [a, b, z, c, d]$$

On peut retrancher un élément d'un ensemble, ou en ajouter un ou plusieurs :

```
> restart:  e:={1,2,3,5}:  e union {4}, e minus {2}, {op(e),a,b};
```

$$\{1, 2, 3, 4, 5\}, \{1, 3, 5\}, \{1, 2, 3, 5, a, b\}$$

Ici, on supprime le troisième élément de la séquence s :

```
> restart:  s:=a,b,c,d:  op(subsop(3=NULL,[s]));
```

$$a, b, d$$

On peut également tester la présence d'un élément dans une séquence :

```
> restart:  s:=a,b,c,d:  member(d,[s],'pos'), pos;
```

$$true, 4$$

2.8.7. Mapper des opérations sur des ensembles ou des listes

Voici deux fonctions d'une variable, mappées sur la même liste :

```
> L:=[1,2,3,4]:  map(sqrt,L), map(x->1/x,L);
```

$$[1, \sqrt{2}, \sqrt{3}, 2], [1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}]$$

Ici, on mappe une fonction de deux variable sur la liste L .

Le premier argument prend comme valeurs successives les différents éléments de L .

Le deuxième argument est constant et précisé à la fin de l'instruction `map` :

```
> restart:  L:=[x,y,z,t,u]:  map((a,b)->a^b,L,3);
```

$$[x^3, y^3, z^3, t^3, u^3]$$

On élève au carré les éléments d'un ensemble.

Dans le résultat, les doublons sont éliminés :

```
> map(x->x^2,{-4,-3,-1,0,1,2,4});
```

$$\{16, 0, 1, 4, 9\}$$

Voici une fonction de trois variables mappée sur un ensemble.

Les deux arguments supplémentaires sont précisés en fin d'instruction :

```
> restart:  map((x,y,z)->y+x*z,{a,b,c,d},2,3);
```

$$\{2 + 3c, 2 + 3d, 2 + 3a, 2 + 3b\}$$

Voici comment additionner ou multiplier terme à terme les éléments de deux listes.

Dans les deux cas les deux listes sont de longueur différente.

- Dans le premier cas, la liste la plus longue est tronquée.
- Dans le deuxième cas, l'argument optionnel sert à remplacer le ou les éléments manquants de la liste la plus courte.

```
> restart:  zip('+',[a,b,c],[x,y,z,t]); zip('*',[a,b],[x,y,z],3);
```

$$[a + x, b + y, c + z]$$

$$[a x, b y, 3 z]$$

Ici on applique la fonction `min` aux éléments de deux listes de même longueur :

```
> zip(min,[5,3,8,4,9],[2,5,7,5,1]);
```

$$[2, 3, 7, 4, 1]$$

2.9. TABLES

2.9.1. Noms indicés

Un nom indicé est un objet s'écrivant $name[index]$, où $name$ est un nom et $index$ une expression ou une séquence d'expressions.

Par exemple $a[1]$, $b[2, 5, 3]$, $xyz[1, u + v]$ sont des noms indicés.

Le nom $name$ peut lui-même être un nom indicé. Exemples : $a[x][3, 5]$ ou $ab[x, y][z]$.

Quand un nom indicé $name[index]$ est évalué, $name$ et $index$ sont évalués. Si $index$ s'évalue sur un entier n et si $name$ conduit à une séquence, un ensemble, ou une liste, alors $name[n]$ donne l'élément de position n dans cette séquence, cet ensemble, ou cette liste.

Si le nom $name$ n'est pas assigné, l'évaluation de $name[index]$ (donc en fait de $index$) conduit encore à un nom indicé, utilisable comme opérande dans n'importe quelle expression.

Il existe une autre structure permettant de grouper des objets et d'y accéder par l'intermédiaire d'indices. Ce sont les *tables*, dont des spécialisations sont les structures de type *tableau*, *matrice* ou *vecteur*.

Un des intérêts des tables est que les indices peuvent être quelconques, et pas uniquement des entiers consécutifs de 1 à n comme dans les séquences, ensembles ou listes.

2.9.2. Création d'une table

Il y a deux manières de créer un objet de type *table* :

- Une manière *explicite*, en utilisant l'instruction **table**.
- Une manière *implicite* en assignant un nom indicé.

Voici d'abord la manière explicite :

- **table()** crée un objet de type *table*, ne contenant pour l'instant aucune entrée.
- **table(list)** crée un objet de type *table*, initialisé d'après le contenu de l'objet *list*.

Si tous les éléments de *list* sont des égalités, alors chaque égalité $index = entry$ fait de *entry* un élément de la table, accessible au moyen du champ *index* correspondant. Si *index* ou *entry* sont des séquences, elles doivent être encadrées par les caractères (et).

Si les éléments de $list = [elt_1, elt_2, \dots, elt_n]$ ne sont pas tous des équations, alors *list* est considérée comme s'écrivant $[1 = elt_1, 2 = elt_2, \dots, n = elt_n]$ et ce qui précède s'applique (les indices sont donc $1, 2, \dots, n$).

On mémorise la table dans un nom en écrivant $name := \mathbf{table}()$ ou $name := \mathbf{table}(list)$. Voici maintenant la manière *implicite* de créer une table.

Considérons un nom indicé $name[index]$ où le nom $name$ n'est pas assigné.

L'instruction $\mathbf{name}[index] := entry$ crée une table nommée $name$, ne comportant au départ qu'un seul élément (la valeur de *entry*), accessible au moyen de l'indice *index*.

2.9.3. Ajouter ou supprimer des éléments à une table

Soit *name* le nom d'une table déjà créée.

- Si *index* n'est pas l'indice (ou la séquence d'indices) permettant d'accéder à un élément existant de la table, alors l'instruction *name[index] := entry*, ajoute l'élément *entry*, le rendant accessible au moyen de l'indice (ou de la séquence d'indices) *index*.
- Si au contraire *index* permet déjà d'accéder à un élément existant de la table, alors l'instruction précédente effectue le remplacement de l'ancien élément par le nouveau.
- On peut notamment effacer une entrée par : *name[index] := 'name'[index]*.

On notera bien les apostrophes. La syntaxe est analogue à celle qui permet de *désassigner* n'importe quel nom.

2.9.4. Lecture et affichage d'une table

Si *name* est le nom d'une table, et si *index* est un indice (ou une séquence d'indices) permettant d'accéder à un élément de la table, alors l'évaluation de *name[index]* produit cet élément.

Avec les mêmes notations, **indices**(*name*) donne la séquence formée des listes [*index*], pour tous les indices *index* permettant d'accéder aux différents éléments de la table.

L'utilisation de listes est nécessaire car un indice peut lui-même être une séquence.

De même, **entries**(*name*) donne la séquence formée des listes [*entry*], pour chacun des éléments *entry* connus de la table.

On ne peut pas prévoir à priori l'ordre dans lequel apparaissent les éléments des séquences produites par **indices** et **entries**, mais les deux ordres sont cohérents l'un avec l'autre.

L'évaluation des *tables* obéit à une règle particulière, dite *évaluation vers le dernier nom* (ce comportement sera aussi celui des *procédures*).

Si par exemple *name* est le nom d'une table, et si on a effectué les assignations *name2:=name* puis *name3:=name2*, alors l'évaluation du nom *name3* (ou celle des noms *name2* et *name*) s'arrêtera au nom *name*.

Pour réellement évaluer le nom *name* en le contenu de la table, il faudra utiliser la syntaxe **eval**(*name*), **op**(*name*), ou **print**(*name*) (cette dernière uniquement pour affichage).

On pourra aussi afficher le contenu d'une table *name* par **lprint**(**eval**(*name*)): comme toujours avec **lprint**, l'affichage est effectué d'une façon linéaire (pas de *pretty print*).

2.9.5. Dupliquer une table

Supposons que *a* soit le nom d'une table et que l'entrée *a*[4, 5] soit définie et égale à 2.

Si on exécute l'instruction *b:=a*, le contenu de *b* est le nom *a*, et pas la table désignée par *a* : c'est une conséquence de l'évaluation vers le dernier nom.

La valeur de *b*[4, 5] est bien sûr 2, mais parce que *b*[4, 5] s'évalue d'abord en *a*[4, 5], puis en 2.

En particulier, si on écrit *a*[4, 5] := 0, alors l'évaluation de *b*[4, 5] donnera 0, car *b*[4, 5] est automatiquement évalué d'abord en *a*[4, 5].

De même si on pose $b[4, 5] := 3$, alors l'évaluation de $a[4, 5]$ donnera 3.

En effet l'instruction $b[4, 5] := 3$ est d'abord évaluée en $a[4, 5] := 3$ avant que l'assignation ne soit réalisée.

On peut résumer cette situation en disant que a et b pointent vers la même table.

Si on veut *dupliquer* la table désignée par a en une nouvelle table nommée b , identique au départ à la précédente, mais autonome de celle-ci, on utilisera la syntaxe $b := \text{copy}(a)$.

2.9.6. Fonctions d'indexation

Il est possible de définir des tables ayant des propriétés particulières. Une table peut ainsi être déclarée *symétrique*, *antisymétrique*, *creuse*, *diagonale* ou *identité*.

Cela se décide à la création de la table en utilisant l'un des mots réservés suivants : **symmetric**, **antisymmetric**, **sparse**, **diagonal** ou **identity**.

Pour sélectionner l'une de ces *fonctions d'indexation*, on précisera le mot correspondant comme paramètre de l'instruction **table**, avant ou après la liste des initialisations.

- Pour une table *creuse* (**sparse**) :

Les entrées non encore précisées dans la table sont supposées valoir 0.

- Pour une table *symétrique* (**symmetric**) :

Deux séquences d'indices se déduisant l'une de l'autre par simple permutation conduisent à des entrées identiques. Cela signifie par exemple que si on pose $\text{name}[1, 2] := 4$, alors implicitement on a aussi posé $\text{name}[2, 1] := 4$.

- Pour une table *antisymétrique* (**antisymmetric**) :

Les entrées correspondant à deux listes d'indices se déduisant l'une de l'autre par une permutation sont réputées égales si cette permutation est paire (c'est-à-dire se décompose en un nombre pair d'échanges) et opposées sinon.

Si on pose par exemple $\text{name}[1, 2, 3] := x$, cela signifie qu'implicitement on a aussi posé $\text{name}[2, 1, 3] := -x$ et $\text{name}[2, 3, 1] := x$.

- Pour une table *diagonale* :

Les entrées relatives à des séquences d'indices non tous égaux sont nulles.

- Pour une table *identité* :

C'est la même chose, mais les entrées relatives à des séquences d'indices tous égaux sont aussi réputées être égales à 1.

On peut définir ses propres fonctions d'indexation, mais cela relève d'une utilisation avancée de Maple. Enfin toute table possède une fonction d'indexation : par défaut, il s'agit de l'objet NULL (séquence vide).

2.9.7. Opérandes d'une table

Une *table* a deux opérandes : la fonction **nops** renvoie donc la valeur 2.

- Le premier, **op**(1,...), est la fonction d'indexation (par défaut, la séquence vide **NULL**).
- Le second, **op**(2,...), est la liste des égalités *index* = *entry* (chaque fois que *index* est une séquence, elle est encadrée par deux parenthèses).
- L'expression **op**(0,...) renvoie le nom **table**.

2.10. TABLES : EXEMPLES

2.10.1. Noms indicés

On définit une liste L . Le quatrième élément de L est lui-même une liste, $[d, e, [f, g], h]$.

$L[4][3]$ extrait le troisième élément de celle-ci, soit la liste $[f, g]$.

Enfin $L[4][3][1]$ extrait le premier élément de $[f, g]$, c'est-à-dire f .

```
> restart:  L:=[a,b,c,[d,e,[f,g],h],i]; L[1]; L[4]; L[4][3]; L[4][3][1];
```

$$L := [a, b, c, [d, e, [f, g], h], i]$$

$$a$$

$$[d, e, [f, g], h]$$

$$[f, g]$$

$$f$$

On voit ici comment effectuer des calculs sur des noms indicés :

```
> restart:  b:=a:  a[1]+a[x]+3*a[2]; a[1,2]+b[3]+b[1,2];
```

$$a_1 + a_x + 3a_2$$

$$2a_{1,2} + a_3$$

2.10.2. Création d'une table

On crée la table t d'une façon explicite.

On précise pour cela la liste $[a, b, c]$ de ses trois éléments.

Par défaut, les trois indices sont respectivement 1, 2 et 3 :

```
> restart:  t:=table([a,b,c]); indices(t); entries(t);
```

```
t:=table([
```

```
2 = b
```

```
3 = c
```

```
1 = a
```

```
])
```

$$[2], [3], [1]$$

$$[b], [c], [a]$$

Voici une autre manière de créer une table t .

On précise par exemple que x est l'élément de la table pour l'indice $(a, 2)$.

On peut ensuite évaluer $t[a, 2]$: on retrouve x .

En revanche, $t[1, 3]$ reste inchangé car on n'a rien placé dans t à cette "adresse".

```
> restart:  t:=table([(a,2)=x,(2,b,5)=z]); t[a,2], t[1,3], t[2,b,5];
```

```
t:=table([
(a, 2) = x
(2, b, 5) = z
])
```

$x, t_{1,3}, z$

A la suite de la ligne précédente, on cherche à afficher la table t .

Evaluer le nom t ne donne rien : l'évaluation reste bloquée sur le dernier nom.

Pour afficher le contenu de t , on utilisera en général `eval(t)`.

L'instruction `lprint(eval(t))` provoque un affichage en mode *linéaire* (et comme `print`, cette instruction n'agit pas sur l'historique des calculs, c'est-à-dire sur le contenu de %).

```
> t; eval(t); lprint(eval(t));
```

t

```
table([
(a, 2) = x
(2, b, 5) = z
])
```

```
table([(a, 2)=x,(2, b, 5)=z])
```

Voici la séquence des indices de la table t , puis celle de ses éléments :

```
> indices(t); entries(t);
```

$[a, 2], [2, b, 5]$

$[x], [z]$

On voit comment vider le terme d'indice $(a, 2)$ de la table t :

```
> t[a,2]:='t[a,2]':  op(t);
```

```
table([
(2, b, 5) = z
])
```

On crée une table *tab* de manière implicite, en définissant l'élément d'indice (1, 2, 3) :

```
> restart:  tab[1,2,3]:=5!; eval(tab);
```

$$tab_{1,2,3} := 120$$

```
table([
(1, 2, 3) = 120
])
```

2.10.3. Évaluation vers le dernier nom

On crée une table, qu'on place dans *t1*.

On place ensuite le nom *t1* dans la variable *t2*, puis le nom *t2* dans *t3*.

On voit que l'évaluation de *t1*, *t2* et *t3*, conduit à chaque fois à *t1*.

Pour afficher le contenu de *t3*, il faut utiliser l'instruction `eval` :

```
> restart:  t1:=table([a,b,c]):  t2:=t1:  t3:=t2:  t1,t2,t3; eval(t3);
```

$$t1, t1, t1$$

```
table([
2 = b
3 = c
1 = a
])
```

A la suite de la ligne précédente, on supprime l'élément d'indice 2 de la table *t3*.

On crée ensuite un élément d'indice 4 et de contenu *z*.

En fait ces affectations rejaillissent sur la table contenue dans *t1*, vers laquelle pointent les variables *t2* et *t3*.

```
> t3[2]:='t3[2]':  t2[4]:=z:  t1, t2, t3; eval(t1);
```

$$t1, t1, t1$$

```
table([
3 = c
4 = z
1 = a
])
```

A la suite de la ligne précédente, on duplique la table *t1* et on place le résultat dans *t2*.

On duplique la table contenue dans *t2* et on place le résultat dans *t3*. A ce moment les trois tables sont identiques, mais elles sont devenues indépendantes l'une de l'autre.

On le voit par exemple avec la suppression de $t2[4]$ et la création de $t3[5]$.

```
> t2:=copy(t1): t3:=copy(t2): t2[4]:='t2[4]': t3[5]:=xx:
  lprint(eval(t1)); lprint(eval(t2));lprint(eval(t3));

table([(3)=c,(4)=z,(1)=a])
table([(3)=c,(1)=a])
table([(3)=c,(4)=z,(1)=a,(5)=xx])
```

2.10.4. Fonctions d'indexation

On crée une table en lui donnant la fonction d'indexation `symmetric`.

On voit par exemple comment l'évaluation des trois éléments $t[1,2,3]$, $t[2,3,1]$ et $t[3,1,2]$ est redirigée vers le même élément :

```
> t:=table(symmetric): t[1,2,3], t[2,3,1], t[3,1,2];
```

$$t_{1,2,3}, t_{1,2,3}, t_{1,2,3}$$

Voici un exemple analogue, mais avec une table *antisymétrique* :

```
> t:=table(antisymmetric): t[1,2,3], t[2,1,3], t[3,2,1], t[2,3,1];
```

$$t_{1,2,3}, -t_{1,2,3}, -t_{1,2,3}, t_{1,2,3}$$

Quand on définit une table antisymétrique, et qu'on crée par exemple l'élément d'indice $(1,2)$, on crée automatiquement celui d'indice $(2,1)$, et les deux valeurs sont opposées :

```
> restart: t:=table(antisymmetric,[(1,2)=xx]): t[2,1];
```

$$-xx$$

Quand on crée une table de type *sparse*, on peut très bien entrer telle ou telle valeur.

Les entrées non définies par l'utilisateur sont alors considérées comme contenant 0 :

```
> t:=table(sparse,[2=10]): t[1], t[5,2,3], t[4,-1], t[2];
```

$$0, 0, 0, 10$$

Avec une table de type *diagonal*, les coefficients diagonaux sont laissés au libre choix de l'utilisateur, et les autres sont nuls :

```
> t:=table(diagonal): t[1,2,4], t[3,3], t[5,5,5], t[5,5,6];
```

$$0, t_{3,3}, t_{5,5,5}, 0$$

Si une table est déclarée diagonale, on ne peut évidemment pas forcer un coefficient non diagonal à être non nul...

```
> t:=table(diagonal,[1,2]=5);
Error, invalid parameters for creation of table or array
```

Voici enfin la création d'une table de type *identity* :

```
> t:=table(identity):  t[1,2,4], t[3,3], t[5,5,5], t[5,5,6];

0, 1, 1, 0
```

2.10.5. Opérandes d'une table

Voici la création d'une table de type *sparse* :

```
> restart:  t:=table([(1,2)=x,(2,a,b)=y+1,5=zzz],sparse);

t:=table(sparse, [
(1, 2) = x
5 = zzz
(2, a, b) = y + 1
])
```

On évalue ensuite le nombre des opérandes de cette table.

On voit encore qu'il est nécessaire d'utiliser `eval` pour accéder effectivement à la table :

```
> nops(t), nops(eval(t));
```

1, 2

A la suite des deux lignes précédentes, voici les opérandes d'indice 0 (le type `table`), d'indice 1 (la fonction d'indexation) et d'indice 2 (la liste des égalités *index = entry*) :

```
> op(0,eval(t)); op(1,eval(t)); op(2,eval(t));
```

table

sparse

$[(1, 2) = x, 5 = zzz, (2, a, b) = y + 1]$

2.11. TABLEAUX

2.11.1. Création d'un tableau

Un *tableau* (*array*) est une forme particulière de *table*.

Plus précisément un tableau à n dimensions est une table dont les *index* sont des séquences de n entiers, le k -ième de ces entiers, c'est-à-dire celui qui correspond à la k -ième dimension, ne pouvant prendre que les valeurs d'un intervalle (*range*) spécifique à cette dimension.

On crée un tableau au moyen de l'instruction **array**.

On peut créer un tableau uniquement en précisant ses dimensions, sans fixer la valeur d'aucune entrée pour l'instant. Pour cela on se contente de préciser les *ranges* propres à chaque dimension.

- **array**($m..n$) crée un tableau à une dimension. Les indices permettant d'accéder à ce tableau sont alors les entiers i compris entre les entiers m et n .
- **array**($m..n, p..q$) crée un tableau à deux dimensions. Si on stocke un tel tableau sous le nom *name*, on accédera à ses entrées en écrivant *name*[i, j], où l'entier i est compris entre les entiers m et n , et où j est compris entre p et q .

Plus généralement **array**($range_1, \dots, range_n$), où chaque $range_k$ est un intervalle d'entiers, crée un tableau à n dimensions.

On peut également créer un tableau en précisant uniquement les entrées. Le nombre de dimensions et les *ranges* correspondants s'en déduisent, avec la seule différence que ces intervalles débiteront tous à 1.

- **array**($[e_1, e_2, \dots, e_n]$), où e_1 n'est pas une liste, crée un tableau à une dimension, où les entrées e_1, \dots, e_n correspondront respectivement aux indices $1, \dots, n$.
- **array**($[list_1, list_2, \dots, list_n]$), où $list_1$ est une liste dont les éléments ne sont pas des listes, crée un tableau à deux dimensions, la première étant n , la deuxième étant la longueur de la liste $list_1$.

Les éléments de $list_1$ définissent la première ligne du tableau.

Les listes suivantes $list_2, \dots, list_n$ doivent être de longueur inférieure ou égale à celle de $list_1$ et permettent de définir le début ou la totalité de chacune des $n - 1$ autres lignes du tableau, les entrées manquantes restant indéfinies.

Plus généralement, la syntaxe **array**($[list_1, list_2, \dots, list_n]$) permet de créer et d'initialiser des tableaux de dimension 3 ou supérieure.

Pour un tableau de dimension 3, par exemple, les éléments de $list_1$ doivent être des listes dont les éléments eux-mêmes ne doivent pas être des listes...

On peut créer un tableau en mélangeant les deux syntaxes précédentes (séquence des *ranges* relatifs à chaque dimension et initialisation de tout ou partie des entrées de la table). Les deux types d'indication ne doivent évidemment pas se contredire.

On peut enfin préciser une *fonction d'indexation*, les mêmes que pour les *tables*.

La syntaxe générale de création d'un tableau est donc **array**(*index_func*, *seq_ranges*, *list*), les trois arguments pouvant être donnés dans un ordre quelconque, les *ranges* devant figurer consécutivement dans une séquence.

2.11.2. Lecture et affichage d'un tableau

Contrairement aux tables, on ne peut pas ajouter ou supprimer un élément à un tableau, puisque les dimensions de celui-ci sont fixées à sa création.

On peut simplement modifier une entrée existante, la définir si ce n'était déjà fait, ou la rendre indéfinie en la désassignant.

Pour les objets de type *array*, comme pour les objets de type *table*, les fonctions **indices** et **entries** donnent la liste de tous les *index* et de toutes les *entrées*, en se limitant aux entrées et aux indices réellement définis.

On rappelle que l'ordre dans lequel sont donnés ces *index* et ces *entrées* est imprévisible, mais que les deux ordres sont cohérents l'un avec l'autre.

Les tableaux, comme les tables, sont soumis à la règle de l'évaluation vers le dernier nom.

Si on a mémorisé un tableau sous le nom *name*, il faudra écrire **eval**(*name*) ou **op**(*name*) pour que l'évaluation conduise réellement au tableau.

Pour l'affichage on pourra aussi utiliser **print**(*name*).

L'instruction **copy** permet de créer une copie d'un tableau, indépendante de l'original.

Les tableaux à une ou à deux dimensions sont affichés sous une forme matricielle traditionnelle, à la condition cependant que les *ranges* relatifs à chaque dimension commencent par 1. On parlera alors de *vecteur* en dimension 1 (*vector*) ou de *matrice* en dimension 2 (*matrix*).

Le package **linalg** contient des instructions permettant de créer plus simplement encore des vecteurs ou des matrices.

Sinon, et en particulier à partir de 3 dimensions, les tableaux sont affichés comme sont le sont les tables, à raison d'une égalité *index* = *entry* par ligne.

2.11.3. Opérandes d'un tableau

Contrairement aux objets de type *table*, qui n'ont que deux opérandes, les objets de type *array* en ont trois.

- Le premier, **op**(1, ...), est le nom de la fonction d'indexation.

Par défaut, c'est la séquence vide **NULL**.

- Le second, **op**(2, ...), est la séquence des *ranges* relatifs à chaque dimension.
- Le troisième, **op**(3, ...), est la liste des égalités *index* = *entry*.

Chaque fois que *index* est une séquence, elle est encadrée par deux parenthèses.

Pour un tableau *tab*, l'expression **op**(0, *tab*) renvoie le nom **array**.

2.11.4. Conversions entre listes, tables et tableaux

Il y a peu de différences apparemment entre une liste et un tableau à une dimension, et notamment quand ce tableau est un *vecteur*, c'est-à-dire quand l'intervalle des indices débute à 1, car les deux objets sont affichés exactement de la même manière.

Mais ces objets sont différents quant à la façon dont ils sont codés en mémoire et aux fonctions qui leurs sont applicables.

Par exemple l'instruction `sort` s'applique à une liste et pas à un vecteur.

Il peut donc être utile de transformer une liste en vecteur, ou réciproquement.

On dispose des possibilités suivantes de conversion :

- `convert(list, array)`

Convertit la liste *list* en un *vecteur*.

Ici le premier élément de *list* ne doit pas être lui-même une liste.

- `convert(vect, list)`

Convertit le tableau *vect*, de dimension 1, en la liste de ses éléments.

- `convert([list1, ..., listn], array)`

Convertit une liste de listes en une *matrice*.

Ici le premier élément de *list₁* ne doit pas être une liste.

Le nombre *n* donne la première dimension (le *range* est 1..*n*) et la longueur *p* de *list₁* donne la deuxième dimension (le *range* est 1..*p*).

Les listes suivantes doivent être de longueur au plus égale à celle de *list₁*.

- `convert(matr, listlist)`

Convertit une matrice, ou un tableau de dimension *i* 1, en une liste de listes.

- `convert(tab, array, m..n)`

Convertit la table *tab* en un tableau de dimension 1, avec un range *m..n*.

- `convert(tab, array, m..n, p..q)`

Convertit la table *tab* en un tableau de dimension 2, avec les ranges *m..n* et *p..q*.

2.11.5. Opérations sur vecteurs et matrices

On peut effectuer les opérations arithmétiques classiques (somme, produit, inverse, etc.) sur des *vecteurs* ou des *matrices*, c'est-à-dire des objets de type *array* de dimension 1 ou 2 avec des intervalles d'indice débutant à 1.

Les opérateurs sont `+`, `-`, `&*`, et `^`. Pour être interprétée comme un calcul matriciel à effectuer, l'expression doit être évaluée par `evalm` (littéralement *evaluate to matrix*).

L'opérateur `*` peut être utilisé mais avec prudence, notamment à cause des simplifications automatiques dues au fait que cet opérateur est considéré comme commutatif par Maple.

Toujours à condition d'utiliser `evalm`, on peut multiplier une matrice par une constante, ou plus généralement par une expression, mais cette fois-ci en utilisant l'opérateur `*`.

Dans les calculs matriciels les *vecteurs* sont considérés comme des *matrices colonnes*.

On peut calculer l'inverse d'une matrice A par `evalm(A^(-1))`.

On peut calculer sa puissance n -ième par `evalm(A^n)`.

La matrice identité peut être désignée par `&*()`.

Dans une somme, un opérande non matriciel sera considéré comme multiple de l'identité.

Enfin `evalm` permet de mapper une fonction f sur une matrice ou un vecteur *matr*.

Pour cela, la syntaxe est `evalm(f(matr))`.

Notons que `map` est encore utilisable sur les *tables*, et donc sur les objets de type *array*, la syntaxe étant `map(f, tab)` si f ne prend qu'un argument et `map(f, tab, arg2, \dots, arg_n)` si f prend n arguments (le premier d'entre eux étant pris dans la table et les suivants étant fixés aux valeurs $arg2, \dots, arg_n$ indiquées dans l'instruction.)

2.12. TABLEAUX : EXEMPLES

2.12.1. Création, affichage, lecture et opérandes d'un tableau

On crée un tableau à une seule dimension, l'intervalle des indices étant 0..2 :

```
> v:=array(0..2); lprint(eval(v)); print(v);
```

$$v := \text{array}(0..2, [])$$

```
array(0 .. 2, [])
```

```
array(0..2, [
(0) = v0
(1) = v1
(2) = v2
])
```

Voici maintenant un tableau à deux dimensions, avec des indices de 1 à 2 pour la première dimension, de de -3 à -1 pour la seconde :

```
> t:=array (1..2,-3..-1); print(t);
```

$$t := \text{array}(1..2, -3..-1, [])$$

```
array(1..2, -3..-1, [
(1, -3) = t1,-3
(1, -2) = t1,-2
(1, -1) = t1,-1
(2, -3) = t2,-3
(2, -2) = t2,-2
(2, -1) = t2,-1
])
```

Dans le tableau précédent, on initialise deux entrées.

```
> t[1,-2]:=x+1: t[1,-3]:=10: lprint(eval(t)); indices(t); entries(t);
```

```
array(1 .. 2, -3 .. -1, [(1, -2)=x+1, (1, -3)=10])
```

$$[1, -2], [1, -3]$$

$$[x + 1], [10]$$

Toujours pour le tableau précédent, on évalue l'entrée d'indices (1, -2).

Puis on désassigne cette entrée avant d'en initialiser une autre :

```
> t[1,-2]; t[1,-2]:='t[1,-2]': t[2,-3]:=99: lprint(eval(t));
```

$$x + 1$$

```
array(1 .. 2, -3 .. -1, [(2, -3)=99, (1, -3)=10])
```

On crée maintenant un tableau à trois dimensions :

```
> t:=array(1..2,1..4,1..3): t[2,3,1]:=999: lprint(eval(t));  
array(1 .. 2, 1 .. 4, 1 .. 3,[(2, 3, 1)=999])
```

Le tableau v , à une dimension, initialisé par la liste de ses éléments, est un *vecteur* :

```
> restart: v:=array([a,a+1,b,a*b]); indices(v); entries(v);
```

$$v := [a, a + 1, b, a b]$$

$$[1], [2], [3], [4]$$

$$[a], [a + 1], [b], [a b]$$

Voici plusieurs manières d'afficher le vecteur précédent :

```
> eval(v); op(v); print(v); lprint(eval(v)):
```

$$[a, a + 1, b, a b]$$

$$[a, a + 1, b, a b]$$

$$[a, a + 1, b, a b]$$

```
array(1 .. 4,[(1)=a,(2)=a+1,(3)=b,(4)=a*b])
```

On crée une matrice en précisant ses éléments, mais de manière incomplète :

```
> restart: m:=array([[a,b,c],[d],[e,f]]); lprint(eval(m));
```

$$m := \begin{bmatrix} a & b & c \\ d & m_{2,2} & m_{2,3} \\ e & f & m_{3,3} \end{bmatrix}$$

```
array(1..3,1..3,[(3,1)=f,(1,2)=b,(2,2)=e,(1,3)=c,(1,1)=a,(2,1)=d])
```

Voici une autre matrice, initialisée complètement par la donnée de ses éléments :

```
> restart: m:=array([[a+b,c+d,e+f,g+h],[a-c,b-d,e-g,f-g]]);
```

$$m := \begin{bmatrix} a + b & c + d & e + f & g + h \\ a - c & b - d & e - g & f - g \end{bmatrix}$$

Voici un tableau à 3 dimensions, initialisé en partie par la donnée de ses éléments :

```
> restart:  m:=array([ [ [1,5,2],[3,2] ],[ [4,8,3],[5] ]]);
```

```
m := array(1..2, 1..2, 1..3, [
(1, 1, 1) = 1
(1, 1, 2) = 5
(1, 1, 3) = 2
(1, 2, 1) = 3
(1, 2, 2) = 2
(1, 2, 3) = m1,2,3
(2, 1, 1) = 4
(2, 1, 2) = 8
(2, 1, 3) = 3
(2, 2, 1) = 5
(2, 2, 2) = m2,2,2
(2, 2, 3) = m2,2,3
])
```

La matrice m est créée en définissant à la fois ses dimensions et certains de ses éléments :

```
> restart:  m:=array(1..3,1..4,[[a],[b,c],[d,e,f]]);
```

$$m := \begin{bmatrix} a & m_{1,2} & m_{1,3} & m_{1,4} \\ b & c & m_{2,3} & m_{2,4} \\ d & e & f & m_{3,4} \end{bmatrix}$$

On définit ici une matrice symétrique. On voit comment, même encore indéfinis, deux coefficients symétriques l'un de l'autre par rapport à la diagonale sont égaux.

```
> m:=array(symmetric,1..3,1..3):  print(m);
```

$$\begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} \\ m_{1,2} & m_{2,2} & m_{2,3} \\ m_{1,3} & m_{2,3} & m_{3,3} \end{bmatrix}$$

Dans la matrice précédente, on définit quelques entrées :

```
> m[1,2]:=10:  m[2,3]:=100:  m[3,3]:=999:  print(m);
```

$$\begin{bmatrix} m_{1,1} & 10 & m_{1,3} \\ 10 & m_{2,2} & 100 \\ m_{1,3} & 100 & 999 \end{bmatrix}$$

Voici alors les opérandes de la matrice obtenue :

```
> op(0,eval(m)), op(1,eval(m)); op(2,eval(m)); op(3,eval(m));
```

array, symmetric

1..3, 1..3

[(2, 3) = 100, (1, 2) = 10, (3, 3) = 999]

Voici enfin les indices et les entrées de cette matrice :

```
> indices(m); entries(m);
```

[2, 3], [1, 2], [3, 3]

[100], [10], [999]

On crée maintenant une matrice antisymétrique :

```
> m:=array(1..4,1..4,antisymmetric): m[2,3]:=100: print(m);
```

$$\begin{bmatrix} 0 & m_{1,2} & m_{1,3} & m_{1,4} \\ -m_{1,2} & 0 & 100 & m_{2,4} \\ -m_{1,3} & -100 & 0 & m_{3,4} \\ -m_{1,4} & -m_{2,4} & -m_{3,4} & 0 \end{bmatrix}$$

Voici un vecteur *creux*. On ne définit explicitement que la valeur de $v[5]$. Tous les autres éléments sont nuls par défaut :

```
> v:=array(sparse,1..7): v[5]:=999: print(v);
```

[0, 0, 0, 0, 999, 0, 0]

On définit ici un autre vecteur *creux*, initialisé à la fois par la donnée de sa dimension et par ses deux premiers éléments :

```
> v:=array(1..7,[4,9],sparse);
```

$v := [4, 9, 0, 0, 0, 0, 0]$

La matrice m , de taille 3×8 , est également de type *creux* :

```
> m:=array(1..3,1..8,sparse): m[2,6]:=9999: print(m);
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 9999 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

On crée maintenant une matrice de type diagonal :

```
> m:=array(diagonal,1..4,1..4): print(m);
```

$$\begin{bmatrix} m_{1,1} & 0 & 0 & 0 \\ 0 & m_{2,2} & 0 & 0 \\ 0 & 0 & m_{3,3} & 0 \\ 0 & 0 & 0 & m_{4,4} \end{bmatrix}$$

Voici comment former la matrice identité d'ordre 3 :

```
> m:=array(1..3,1..3,identity): eval(m);
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2.12.2. Conversion entre listes, tables et tableaux

Ces deux objets semblent égaux, mais ils ne le sont pas.

L'un est un vecteur, l'autre est une liste. A l'affichage, rien ne les distingue :

```
> a:=array([30,51,27,83]); b:=[30,51,27,83]; evalb(eval(a)=b);
```

$a := [30, 51, 27, 83]$

$b := [30, 51, 27, 83]$

false

L'instruction `whattype` permet de les différencier, tout comme la fonction `op` qui donne ici la liste de leurs opérandes :

```
> whattype(a), whattype(eval(a)), whattype(b); op(a); op(b);
```

string, array, list

$[30, 51, 27, 83]$

30, 51, 27, 83

On voit que a et b sont codés de manières très différentes en mémoire :

```
> op(2,eval(a)); op(2,b); op(3,eval(a)); op(3,b);
```

1..4

51

$[1 = 30, 2 = 51, 3 = 27, 4 = 83]$

27

Autre différence, on peut trier la liste b , pas le vecteur a :

```
> sort(b), sort(a); sort(eval(a));
```

$[[1, 7, 5], [3, 9, 2]], a$

Error, invalid argument for indets

Voici comment convertir le vecteur a en une liste, et la liste b en un vecteur.

On notera bien le résultat du deuxième test, car a et bb sont bien égaux :

```
> aa:=convert(a,list); bb:=convert(b,array);
```

```
evalb(aa=b), evalb(bb=a);
```

```
lprint(eval(bb)); lprint(eval(a));
```

$aa := [30, 51, 27, 83]$

$bb := [30, 51, 27, 83]$

$true, false$

```
array(1 .. 4, [(1)=30, (2)=51, (3)=27, (4)=83])
```

```
array(1 .. 4, [(1)=30, (2)=51, (3)=27, (4)=83])
```

En fait `evalb(...=...)` ne permet pas de vérifier l'égalité de deux tableaux pourtant égaux, sauf dans le cas où les deux variables pointent en mémoire sur le même tableau :

```
> v1:=array([4,5]): v2:=v1: v3:=array([4,5]): evalb(v1=v2); evalb(v1=v3);
```

$true$

$false$

Voici comment convertir une liste de listes en matrice :

```
> a:=[[1,7,5],[3,9,2]]; b:=convert(a,array);
```

$a := [[1, 7, 5], [3, 9, 2]]$

$b := \begin{bmatrix} 1 & 7 & 5 \\ 3 & 9 & 2 \end{bmatrix}$

La conversion inverse est également possible :

```
> a:=array([[1,7,5],[3,9,2]]); b:=convert(a,listlist);
```

$a := \begin{bmatrix} 1 & 7 & 5 \\ 3 & 9 & 2 \end{bmatrix}$

$b := [[1, 7, 5], [3, 9, 2]]$

2.12.3. Opérations sur vecteurs et matrices

On crée une matrice carrée M , de dimension 3×3 :

```
> restart:  M:=array([[1,2,lambda],[1,0,-1],[1,1,0]]);
```

$$M := \begin{bmatrix} 1 & 2 & \lambda \\ 1 & 0 & -1 \\ 1 & 1 & 0 \end{bmatrix}$$

On forme également deux matrices N et P de tailles respectives 3×2 et 2×3 :

```
> N:=array([[1,0],[-1,1],[2,1]]); P:=array([[1,1,2],[-1,3,2]]);
```

$$N := \begin{bmatrix} 1 & 0 \\ -1 & 1 \\ 2 & 1 \end{bmatrix}$$

$$P := \begin{bmatrix} 1 & 1 & 2 \\ -1 & 3 & 2 \end{bmatrix}$$

Voici enfin un vecteur V , de longueur 3 :

```
> V:=array([3,2,4]);
```

$$V := [3, 2, 4]$$

On effectue le produit de la matrice P par le vecteur V .
Ce dernier est alors considéré comme une matrice-colonne :

```
> P*V, P&*V, evalm(P&*V);
```

$$PV, P \& * V, [13, 11]$$

Voici le produit de N par P , mémorisé dans la variable Q :

```
> Q:=evalm(N&*P);
```

$$Q := \begin{bmatrix} 1 & 1 & 2 \\ -2 & 2 & 0 \\ 1 & 5 & 6 \end{bmatrix}$$

On effectue également le produit de P par N :

```
> evalm(P&*N);
```

$$\begin{bmatrix} 4 & 3 \\ 0 & 5 \end{bmatrix}$$

On n'utilisera pas l'opérateur $*$ pour les produits de matrices!

On voit bien pourquoi sur l'exemple suivant. En effet l'expression $PN - NP$ n'a aucun sens mais elle est pourtant automatiquement simplifiée en 0, car Maple considère avant toute chose que l'opérateur $*$ est commutatif :

```
> P*N-N*P;
```

0

L'utilisation de la bonne syntaxe montre l'incompatibilité des dimensions :

```
> evalm(P&*N-N&*P);
```

```
Error, (in linalg[matadd]) matrix dimensions incompatible
```

Les produits MQ et QM existent tous deux mais sont distincts.

```
> M*Q-Q*M,
  evalm(M&*Q),
  evalm(Q&*M);
  evalm(M&*Q-Q&*M);
```

$$0, \begin{bmatrix} -3 + \lambda & 5 + 5\lambda & 2 + 6\lambda \\ 0 & -4 & -4 \\ -1 & 3 & 2 \end{bmatrix}$$

$$\begin{bmatrix} 4 & 4 & \lambda - 1 \\ 0 & -4 & -2\lambda - 2 \\ 12 & 8 & \lambda - 5 \end{bmatrix}, \begin{bmatrix} -7 + \lambda & 1 + 5\lambda & 3 + 5\lambda \\ 0 & 0 & -2 + 2\lambda \\ -13 & -5 & 7 - \lambda \end{bmatrix}$$

Voici l'inverse de la matrice carrée M (dans le cas général, c'est-à-dire $\lambda \neq 1$) :

```
> evalm(M^(-1));
```

$$\begin{bmatrix} \frac{1}{\lambda - 1} & \frac{\lambda}{\lambda - 1} & -\frac{2}{\lambda - 1} \\ -\frac{1}{\lambda - 1} & -\frac{\lambda}{\lambda - 1} & \frac{1 + \lambda}{\lambda - 1} \\ \frac{1}{\lambda - 1} & \frac{1}{\lambda - 1} & -\frac{2}{\lambda - 1} \end{bmatrix}$$

On calcule maintenant la puissance quinzième de la matrice Q :

```
> evalm(Q^15);
```

$$\begin{bmatrix} -17236805051 & 52784156977 & 35547351926 \\ 11133289426 & -34473610102 & -23340320676 \\ -40577125727 & 123878860829 & 83301735102 \end{bmatrix}$$

Voici comment effectuer le produit ou la somme d'une matrice et d'une constante.

Pour la somme, la constante est implicitement multipliée par la matrice identité.

En revanche, le produit s'effectue terme à terme :

```
> evalm(Q+alpha), evalm(alpha*Q);
```

$$\begin{bmatrix} 1+\alpha & 1 & 2 \\ -2 & 2+\alpha & 0 \\ 1 & 5 & 6+\alpha \end{bmatrix}, \begin{bmatrix} \alpha & \alpha & 2\alpha \\ -2\alpha & 2\alpha & 0 \\ \alpha & 5\alpha & 6\alpha \end{bmatrix}$$

On peut *mapper* une fonction quelconque sur les termes d'une table (donc d'un tableau, d'une matrice ou d'un vecteur) au moyen de la fonction `map`.

Pour les matrices et les vecteurs on peut parfois utiliser la syntaxe `evalm(f(matr))`.

Ici on applique les fonctions `sqrt` et `sin` au vecteur V , avec deux syntaxes différentes.

On observera le comportement différent avec `evalm` :

```
> map(sqrt,V), evalm(sqrt(V)); map(sin,V), evalm(sin(V));
```

$$[\sqrt{3}, \sqrt{2}, 2], \sqrt{[3, 2, 4]}, [\sin(3), \sin(2), \sin(4)], [\sin(3), \sin(2), \sin(4)]$$

Voici deux manières d'ajouter la constante α à tous les termes de la matrice Q :

```
> map(x->x+alpha,Q), map((x,y)->x+y,Q,alpha);
```

$$\begin{bmatrix} 1+\alpha & 1+\alpha & 2+\alpha \\ -2+\alpha & 2+\alpha & \alpha \\ 1+\alpha & 5+\alpha & 6+\alpha \end{bmatrix}, \begin{bmatrix} 1+\alpha & 1+\alpha & 2+\alpha \\ -2+\alpha & 2+\alpha & \alpha \\ 1+\alpha & 5+\alpha & 6+\alpha \end{bmatrix}$$

Ici on crée une matrice R , dont les coefficients dépendent de la variable x .

Puis on effectue une dérivation terme à terme :

```
> R:=array([[cos(x),sin(x)],[tan(x),cos(x)]]):
eval(R), map(diff,R,x);
```

$$\begin{bmatrix} \cos(x) & \sin(x) \\ \tan(x) & \cos(x) \end{bmatrix}, \begin{bmatrix} -\sin(x) & \cos(x) \\ 1+\tan(x)^2 & -\sin(x) \end{bmatrix}$$

2.13. LES TYPES DE MAPLE

2.13.1. Types de base

Pour savoir quel est le type d'un objet *obj*, on évaluera l'instruction `whattype(obj)`.

Le résultat est un nom.

On pourra vérifier qu'un objet *obj* est d'un type particulier en évaluant `type(obj,typename)`, où *typename* est l'un des noms prédéfinis par Maple pour représenter tel ou tel type.

Le résultat est `true` ou `false`, suivant que l'objet *obj* est ou n'est pas de type *typename*.

Les types de base de Maple sont :

- `symbol`, `string` : noms symboliques, chaînes de caractères.
- `integer`, `fraction`, `float` : entiers, nombres rationnels, nombres réels.
- `'+'`, `'*'`, `'^'` : sommes, produits, puissances.
- `'='`, `'<'`, `'<='`, `'<>'` : égalités et inégalités.
- `'and'`, `'or'`, `'not'` : expressions booléennes.
- `function` : appels à une fonction, ou composition de fonctions.
- `exprseq`, `set`, `list` : séquences, ensembles, listes.

Remarque : le type `exprseq` ne peut pas être testé par la fonction `type`.

- `indexed`, `table`, `array`, `'..'` : noms indicés, tables, tableaux, intervalles.
- `procedure` : programmes.
- `series` : objets obtenus par l'instruction `series` (développements limités.)
- `uneval` : objets non évalués car encadrés d'apostrophes (`'x'`, `'2+3'`).
- `'.'` : concaténations non évaluées, par exemple `a.(2/3)`.

2.13.2. Réunions ou synonymes de types de base

Certains types sont des regroupements de types de base de Maple. Ils peuvent encore être utilisés comme deuxième argument dans l'instruction `type` (la syntaxe est `type(obj,typename)`), mais on rappelle que cette instruction ne renvoie que des noms de type de base.

Voici une liste non exhaustive des types qui sont réunions de types de base :

- `rational` : regroupe les types `integer` et `fraction`.
- `numeric` : regroupe les types `rational` et `float`.
- `name` : regroupe les types `string` et `indexed`.
- `relation` : regroupe les types `'='`, `'<'`, `'<='`, `'<>'`.

- **logical** : regroupe les types `'and'`, `'or'`, `'not'`.
- **boolean**: regroupe les types `relation` et `logical`.
Ce type reconnaît aussi les constantes `true` et `false`.
- **algebraic** : regroupe en particulier les types `numeric`, `name`, les sommes, produits, puissances, factorielles, et les appels de fonctions non évalués.
- **anything** : regroupe tous les types sauf `exprseq`.

Maple utilise aussi des types synonymes des types de base.

Ces synonymes sont encore utilisables avec l'instruction `type` :

- `'**'` est synonyme du type `'^'`
- `equation` est synonyme du type `'='`
- `range` est synonyme du type `'..'`

2.13.3. Types “précisés”

Alors que les types *réunions* permettent de reconnaître des classes plus importantes d'objets que les types de base (ils *ratissent* plus large), les types *précisés* opèrent des sélections plus fines. En voici une liste non limitative. Ils peuvent être utilisés avec l'instruction `type` :

- **matrix, vector**
Vecteurs ou matrices (dimension 1 ou 2, indices commençant à 1).
- **listlist**
Listes de listes, les listes internes devant avoir même longueur.
- **scalar**
Dans une expression matricielle, tout sauf les vecteurs et les matrices.
- **'!', 'union', 'intersect', 'minus', sqrt**
Les objets résultant de l'appel non évalué à la fonction ou à l'opérateur spécifié.
- **positive, negative, nonneg**
Les nombres positifs (>0), négatifs (<0) ou non négatifs (≥ 0).
- **even, odd**
Les entiers pairs (*even*) ou impairs (*odd*).
- **radical**
Les objets de type puissance (`'^'`), avec un exposant rationnel.
- **mathfunc**
Les noms de fonctions mathématiques connues de Maple (et il y en a beaucoup).

2.13.4. Types rékursifs

Certains types sont dits *rékursifs* car ils supposent une exploration complète de la structure des objets auxquels ils s'appliquent, contrairement aux types précédents qui sont pratiquement tous des *types de surface* car ils n'analysent que l'information située à la racine de l'arborescence de l'expression.

Voici quelques types rékursifs :

- **constant**
Pour vérifier qu'un objet ne contient que des composants constants.
- **polynom**
Les expressions polynômiales par rapport à toutes leurs variables.
- **expanded**
A supposer que l'objet soit un polynôme, vérifie s'il est développé.
- **polynom(*typname*,*name*)**
Les polynômes par rapport à *name*, et dont les coefficients sont de type *typname*.
On utilisera **anything** pour ne poser aucune condition sur les coefficients.
- **polynom(*typname*, {*name*₁, *name*₂, ..., *name*_{*n*}})**
Les polynômes par rapport aux variables *name*₁, *name*₂, ..., *name*_{*n*}.
- **linear, quadratic, cubic, quartic**
Les polynômes de degré total 1, 2, 3 ou 4 par rapport à toutes leurs variables.
On peut utiliser les mêmes options qu'avec le type **polynom**.
- **ratpoly**
Les fractions rationnelles, y compris les polynômes.
Les variantes de syntaxe sont les mêmes qu'avec le type **polynom**.

2.13.5. Types structurés

$[7, -3, 5, 2]$ est visiblement une liste d'entiers : elle est donc de type **list(integer)**.

Ce genre de type connu de Maple est dit *type structuré* car il représente l'image générale de la structure de l'objet auquel on va l'appliquer.

On peut encore utiliser ces noms de type comme deuxième argument de la fonction **type**. Ils permettent en une seule *passé* d'effectuer un contrôle très précis : avec l'exemple précédent, il n'est pas besoin de vérifier que l'objet est de type *list*, puis dans une boucle que les coefficients de cette liste sont de type *integer*.

Le nombre de cas possibles est trop grand pour que tous soient cités ici. On pourra se reporter à l'aide en ligne évaluant la commande **?type[structured]**.

Voici quand même quelques types structurés classiques.

Dans les définitions ci-dessous, typ , typ_1 , typ_2, \dots , typ_n désignent des noms de type.

- $\{typ_1, \dots, typ_n\}$

Tous les objets de l'un au moins des types typ_1 , ou typ_2 , ..., ou typ_n .

- $typ_1 = typ_2$

Les égalités $obj_1 = obj_2$, où obj_1 est de type typ_1 , et obj_2 de type typ_2 .

Il y a une syntaxe analogue avec $<$, $<=$, $>$, $>=$, $<>$, \dots , **and**, **or**, \wedge , $\&+$, $\&*$.

- **set**(typ) , **list**(typ), $'+'(typ)$, $'*(typ)$

Les ensembles, les listes, les sommes ou les produits d'objets du type typ .

- **function**(typ)

Les objets qui sont une fonction $f(obj)$ d'un objet obj du type typ .

On peut même préciser la fonction, et former par exemple le type **exp**(**integer**).

2.13.6. La fonction **hastype**

hastype teste si l'une au moins des sous-expressions de $expr$ est de type $typename$.

La syntaxe est **hastype**($expr, typename$).

La fonction **match** permet également de savoir si une expression obéit à un certain *motif*.

2.14. LES TYPES DE MAPLE : EXEMPLES

2.14.1. Types de base

Voici quelques types numériques :

```
> whattype(1999), whattype(-1), whattype(1/3), whattype(12.345);
```

integer, integer, fraction, float

Le contrôle de type s'effectue après évaluation et/ou simplification automatique.

Ceci explique que $1. - 1$, simplifié en 0, soit de type *integer*.

De même, $2. - 1$ est évalué en 1. (notez le point décimal) et est donc un *float*.

Enfin $\frac{45}{9}$ est simplifié en 5, et est donc de type *integer*.

```
> whattype(1.-1), whattype(2.-1), whattype(45/9);
```

integer, float, integer

Tant qu'un nom de variable est non assigné, il est de type *symbol*.

Sinon il est évalué en son contenu qui peut être une somme, un produit, une puissance, etc.

```
> restart: whattype(x);
```

```
  x:=a+b:  y:=c*d:  z:=e^f:  map(whattype,[x,y,z]);
```

symbol

$[+, *, ^]$

L'expression $a + b * c$ est la somme de a et de $b * c$: elle est donc de type $+$.

En revanche, $(a + b) * c$ est un produit : c'est une expression de type $*$.

$A\&*B$ représente un appel non évalué à la fonction $\&*$.

```
> map(whattype,[a+b*c,(a+b)*c,A&*B]);
```

$[+, *, function]$

On teste ici le type d'expressions qui sont des égalités ou des inégalités :

```
> map(whattype,[a=b,a<>b,a<b,a<=b,a>b,a>=b]);
```

$[=, <>, <, <=, >, >=]$

Voici trois expressions logiques, de types respectifs *and*, *or* et *not*.

```
> map(whattype,[a and b, a or b, not a]);
```

$[and, or, not]$

$[a, b, c]$ est une liste, $\{a, b, c\}$ est un ensemble, et a, b, c est une séquence d'expressions...

```
> whattype([a,b,c]), whattype({a,b,c}), whattype(a,b,c);
```

list, set, exprseq

Quand une variable contient une table (un tableau), il faut comme toujours utiliser `evalm` pour accéder au contenu de cette variable et en tester le type :

```
> x:=table(): y:=array(4..9): map(whattype,[x,y,eval(x),eval(y)]);
```

[symbol, symbol, table, array]

$3..8$ est un intervalle (type *range*), $b[3]$ est un nom indexé, $a.3$ est de type *symbol* (car évalué en le nom $a3$) et $a.(\frac{3}{4})$ est de type *point* car ici la concaténation ne peut pas s'effectuer.

```
> 3..8,b[3],a.3,a.(3/4); map(whattype,[%]);
```

$3..8, b_3, a3, a.(\frac{3}{4})$

[., indexed, symbol, .]

L'expression $'2 + 3'$ s'évalue en 5 et est donc de type *integer*.

En revanche $''2 + 3''$ s'évalue en $'2 + 3'$ et est de type *uneval*.

```
> whattype('2+3'), whattype(''2+3'');
```

integer, uneval

$\exp(1)$ se simplifie en e uniquement pour l'affichage, mais reste considéré comme un appel non évalué à la fonction \exp : c'est donc un objet de type *function*.

Il est de même de $\sin \frac{\pi}{99}$ qui n'est pas simplifié.

En revanche, on lit le type de $\exp(0)$ et $\sin \frac{\pi}{3}$ après simplification.

```
> x:=[exp(1),exp(0),sin(Pi/99),sin(Pi/3)]; map(whattype,x);
```

$x := [e, 1, \sin(\frac{1}{99} \pi), \frac{1}{2} \sqrt{3}]$

*[function, integer, function, *]*

On utilise maintenant `type` pour vérifier si une expression est d'un type particulier.

L'expression $3 * 12$, n'est pas de type $*$, car elle est évaluée en 36.

Pi est une constante symbolique : ce n'est pas un *float*.

L'expression a/b n'est pas de type $/$, tout simplement parce que ce type n'existe pas, mais aussi parce a/b , codé $a * b^{-1}$ en mémoire, est de type $*$.

```
> type(3*12,'*'), type(a*b,'*'), type(Pi,float); type(a/b,'/');
```

false, true, false

```
Error, type '/' does not exist
```

Ici on utilise l'instruction `zip` pour vérifier simultanément si les expressions `15`, `3/2`, `ab` et `1/b` sont respectivement de type *integer*, *fraction*, *symbol* et *fraction*.

```
> zip(type,[15,3/2,ab,1/b],[integer,fraction,symbol,fraction]);
```

[true, true, true, false]

2.14.2. Réunions ou synonymes de types de base

`12345` est un *integer* donc c'est un *rational*. `10/1111` est de type *fraction* donc de type *rational*.

En revanche `12.345` est de type *float*, et pas de type *rational*.

On ne le confondra donc pas avec le rationnel `12345/1000`...

```
> type(12345,rational), type(10/1111,rational), type(12.345,rational);
```

true, true, false

Les trois nombres considérés sont tous de type *numeric*.

```
> map(type,[12345,10/1111,12.345],numeric);
```

[true, true, true]

On voit que tous les objets de la liste suivante sont de type *name* :

```
> restart: map(type,[a,sin,'Maple V',a[1,2],a.3],name);
```

[true, true, true, true, true]

On définit deux listes *x* et *y*, contenant un certain nombre d'expressions logiques :

```
> x:=[a+1=b-2, 2<>3, evalb(2<>3), a<b, not(a<b)];
```

```
y:=[ a and b, a or b, not(a<b), true or a];
```

$$x := [a + 1 = b - 2, 2 \neq 3, \text{true}, a < b, \text{not}(a - b < 0)]$$

$$y := [a \text{ and } b, a \text{ or } b, \text{not}(a - b < 0), \text{true}]$$

Voici quels sont les types des différents éléments de *x* et de *y* :

```
> map(whattype,x), map(whattype,y);
```

[=, <>, symbol, <, not], [and, or, not, symbol]

On teste quels éléments de x sont de type *relation*, et quels éléments de y sont de type *logical* :

```
> map(type,x,relation), map(type,y,logical);
```

```
[true, true, false, true, false], [true, true, true, false]
```

On voit que tous les éléments de x et de y sont de type *boolean* :

```
> map(type,x,boolean), map(type,y,boolean);
```

```
[true, true, true, true, true], [true, true, true, true]
```

Parmi tous ces objets, seuls les deux derniers ne sont pas de type *algebraic* :

```
> map(type,[a,sqrt(a*b),sin(a),a/(a+b),12,sin,{a},table()],algebraic);
```

```
[true, true, true, true, true, true, false, false]
```

Les types `**`, *range* et *equation* sont respectivement synonymes des types `^`, `..` et `=` :

```
> zip(type,[a^b,a=b,4..7],['**',equation,range]);
```

```
[true, true, true]
```

2.14.3. Types précisés

x et y contiennent tous les deux un tableau de dimension 1 :

```
> restart:  x:=array(2..8):  y:=array(1..5):
```

Le contenu de x et y est bien de type *array*.

En revanche seul celui de y est de type *vector* (en effet l'indice de x ne débute pas à 1) :

```
> map(type@eval,[x,y],array), map(type@eval,[x,y],vector);
```

```
[true, true], [false, true]
```

On place maintenant dans x un tableau de dimension 3 et dans y un tableau de dimension 2 :

```
> x:=array(1..3,1..4,1..2):  y:=array(1..5,1..4):
```

Le contenu de x et celui de y sont bien de type *array*.

En revanche seul celui de y est de type *matrix* (les indices du tableau contenu dans x débutent pourtant à 1, mais c'est un tableau de dimension 3) :

```
> map(type,[x,eval(x),eval(y)],array), map(type@eval,[x,y],matrix);
```

```
[true, true, true], [false, true]
```

Dans x , on place une liste de listes. On la convertit en tableau et on place le résultat dans y :

```
> x:=[[1,2,3],[5,6,7]]; y:=array(x);
```

$$x := [[1, 2, 3], [5, 6, 7]]$$

$$y := \begin{bmatrix} 1 & 2 & 3 \\ 5 & 6 & 7 \end{bmatrix}$$

Les instructions suivantes confirment quel est le type du contenu de x et y :

```
> map(whattype,[x,y,eval(y)]), map(type,[x,y],array);
```

$$[list, symbol, array], [false, true]$$

On voit que x est de type *listlist*, mais pas y qui est de type *matrix* :

```
> map(type,[x,y],listlist), map(type,[x,y],matrix);
```

$$[true, false], [false, true]$$

On voit que le type *scalar* recouvre beaucoup de situations différentes :

```
> map(type,[x,y,a+1,exp(3),{a}],scalar);
```

$$[true, false, true, true, true]$$

On place maintenant un ensemble dans x et un autre dans y :

```
> x:={1,2,3}: y:={2,3,4}:
```

L'expression $x \text{ union } y$ n'est pas de type *union* car elle s'évalue en $\{1, 2, 3, 4\}$:

```
> map(type,[x union y, a union b], 'union');
```

$$[false, true]$$

Dans toutes ces expressions, seul 123 est considéré comme de type *positive*.

```
> map(type,[a^2,0,123,abs(a+1)],positive);
```

$$[false, false, true, false]$$

$2 * a + 1$ n'est pas considéré comme de type *odd* (impair), tout simplement parce qu'on ne sait rien de a . En revanche $(8a + 1) \bmod 4$ est de type *odd* car cette expression se simplifie automatiquement en 1...

```
> map(type,[123,2*a+1,0,(8*a+1) mod 4],odd);
```

$$[true, false, false, true]$$

Pour être de type *radical* l'expression doit être une puissance avec exposant rationnel, et par exemple une racine carrée (mais pas `sqrt(16)` qui s'évalue sur l'entier 4) :

```
> map(type,[sqrt(a),a^(1/5),8^(1/3),sqrt(16),a^(b/c)],radical);
```

[true, true, true, false, false]

Les objets de type *mathfunc* sont les noms des fonctions mathématiques usuelles :

```
> map(type,[sin,sin(a),Sin,b,a+b,sqrt,'+'],mathfunc);
```

[true, false, false, false, false, true, false]

2.14.4. Types récursifs

On place dans p et q deux expressions algébriques.

p contient des références à des variables non affectées et n'est donc pas de type *constant*.

Mais q est de ce type car elle s'exprime en fonction de constantes numériques ou symboliques.

La variable r , non assignée, n'est pas considérée comme de type *constant* :

```
> restart:  p:=a+b/3:  q:=exp(2)+Pi/3:  map(type,[p,q,r],constant);
```

[false, true, false]

On place dans p et q deux autres expressions algébriques :

```
> p:=2*x^3+x^2-Pi*x+sin(a):  q:=x^2+(x+y)*z^2:
```

Le contenu de p n'est pas de type *polynom* à cause de la sous-expression $\sin(a)$.

En revanche, p est de type *polynome* en la variable x :

```
> type(p,polynom), type(p,polynom(anything,x));
```

false, true

p est un polynôme en x dont on ne peut pas dire que les coefficients soient de type *numeric*.

Mais on peut dire que ses coefficients sont de type *algebraic* :

```
> type(p,polynom(numeric,x)), type(p,polynom(algebraic,x));
```

false, true

On se souvient que $q = x^2 + (x + y)z^2$: c'est donc un polynôme en toutes ses variables.

C'est même un polynôme en x, y, z à coefficients entiers :

```
> type(q,polynom), type(q,polynom(integer,{x,y,z}));
```

true, true

p est de type *expanded* (développé), mais pas q :

```
> map(type,[p,q],expanded);
```

[true, false]

Ni p ni q ne sont de degré 1 en la variable x .

Seul p est un polynôme de degré 1 en la variable y :

```
> map(type,[p,q],linear(x)); map(type,[p,q],linear(y));
```

[false, false]

[false, true]

p est un polynôme en x de degré 3. Il est donc de type *cubic(x)* :

```
> zip(type,[p,p,p],[quadratic(x),cubic(x),quartic(x)]);
```

[false, true, false]

q est du second degré (donc quadratique) en x , ou par rapport à la paire x, y .

En revanche, le degré total de q par rapport à x, y, z est égal à 3 :

```
> zip(type,[q,q,q],[quadratic(x),quadratic({x,y}),quadratic({x,y,z })]);
```

[true, true, false]

Le quotient $\frac{p}{q}$ n'est pas de type *ratpoly* (fraction rationnelle) par rapport à l'ensemble de ses variables, toujours à cause de la présence de $\sin(a)$.

Mais $\frac{p}{q}$ est de type *ratpoly* par rapport à la variable x :

```
> type(p/q, ratpoly), type(p/q, ratpoly(anything,x));
```

false, true

2.14.5. Types structurés

On place dans x la liste $[3, 8, 2, 4]$.

On voit que Maple la reconnaît en tant que liste d'entiers (attention à la syntaxe) :

```
> restart:  x:=[3,8,2,4]:  type(x,list(integer)), type(x,list(integer));
```

$$false, true$$

On applique la fonction `sqrt` à tous les éléments de x , et on place le résultat dans y .

Les objets de y ne sont pas tous de type *numeric*, mais ils sont tous de type *constant*.

```
> y:=map(sqrt,x); type(y,list(numeric)), type(y,list(constant));
```

$$y := [\sqrt{3}, 2\sqrt{2}, \sqrt{2}, 2]$$

$$false, true$$

On voit que Maple est capable de reconnaître le type “liste de sinus de constantes” :

```
> z:=map(sin,y); type(z,list(sin(constant)));
```

$$z := [\sin(\sqrt{3}), \sin(2\sqrt{2}), \sin(\sqrt{2}), \sin(2)]$$

$$true$$

On voit que la liste $z = [1, a, \frac{2}{3}]$ n'est pas considérée comme une liste d'objets de type *numeric*, mais comme une liste d'objets de type “ou *numeric* ou *name*” :

```
> z:=[1,a,2/3]:  type(z,list(numeric)),type(z,list({numeric,name}));
```

$$false, true$$

Le contenu de la variable z est ici identifié comme un ensemble d'objets de type *algebraic* :

```
> z:={sin(a+1),sqrt(a^2+1),abs(a^3)}; type(z,set(algebraic));
```

$$z := \{\sin(a+1), |a|^3, \sqrt{a^2+1}\}$$

$$true$$

Dans la liste ci-dessous, on trouve les objets qui sont de type *name* ou *integer* :

```
> map(type,[a,12345,3.5,2/3,a[5]],{name,integer});
```

$$[true, true, false, false, true]$$

On voit que Maple est capable de reconnaître une liste de deux éléments dont le premier est de type *integer* et le second de type *range* :

```
> type(a=[12345,5..8],name=[integer,range]);
```

true

Ici on reconnaît une liste comme contenant d'abord le sinus de "n'importe quoi", puis l'image d'un entier par une fonction quelconque :

```
> type([sin(a+1),exp(3)],[sin(anything),anyfunc(integer)]);
```

true

L'expression ci-dessous est un polynôme à coefficients entiers par rapport à $\sin(a)$:

```
> type(sin(a)^5+sin(a)^2+1,polynom(integer,sin(a)));
```

true

Voici comment on peut reconnaître un produit de trois noms :

```
> type(a*b*c,'*(name$3)), type(a*b*c,'&*(name$3));
```

false, true

Le type *identical* permet de contrôler de façon exacte une expression donnée.

Pour cela, Maple considère momentanément cette expression comme un type à part entière.

```
> type(a^3+sin(c),identical(a^3+sin(c)));
```

true

Voici un autre exemple d'utilisation du type *identical* :

```
> type((a+1)^3,identical(a+1)^integer);
```

true

2.14.6. La fonction `hastype`

On place dans x l'expression $1 + \frac{\sin(2c + 1)}{|b|}$.

On voit que cette expression contient un objet de type *integer*, mais pas d'objet de type *float* :

```
> restart:  x:=1+sin(2*c+1)/abs(b):
           hastype(x,integer), hastype(x,float);
```

true, false

x contient une addition, un produit, et le sinus d'un polynôme de degré 1 en la variable c :

```
> hastype(x,'+'), hastype(x,'*'),hastype(x,sin(linear(c)));
```

true, true, true

x ne contient pas d'image d'un *numeric* par une fonction.

En revanche, x contient la valeur absolue d'un nom :

```
> hastype(x,anyfunc(numeric)), hastype(x,abs(name));
```

false, true

2.14.7. Fonction `match` ou fonction `type` ?

La fonction `match` indique que $\sin(x^3)^2$ est bien du type $\sin(x^a)^b$, avec $b = 2$ et $a = 3$.

```
> restart:  match(sin(x^3)^2=sin(x^a)^b,x,'coef'), coef;
```

true, {b = 2, a = 3}

La fonction `type` permet aussi d'identifier le type $\sin(x^{\text{anything}})^{\text{anything}}$:

```
> type(sin(x^3)^2,sin(identical(x)^anything)^anything);
```

true