PETIT COURS DE MAPLE

Paul BARBAROUX

Lycée Clemenceau, Nantes

Ce cours expose les principes de base de Maple, langage de calcul formel utilisé pour l'enseignement d'informatique en classe préparatoire.

On rencontre dans les concours des Grandes Ecoles deux types d'épreuves contenant de l'informatique :

- Dans certains oraux de maths, on demande d'utiliser Maple pour résoudre l'exercice. La partie 1 de ce cours est consacrée au fonctionnement général de Maple et sert essentiellement pour ce premier type d'épreuve. Les fonctions mathématiques proprement dites ne sont pas détaillées ici, *il s'agit juste d'un exposé des commandes de base de Maple. Le logiciel est muni d'une aide très bien faite : ne pas hésiter à en user et en abuser.*

- L'algorithmique *est évaluée à l'écrit des concours d'une part sous la forme d'une épreuve écrite, qui, pour l'instant, n'existe qu'au concours de Polytechnique, et d'utre part sous la forme de questions d'algorithmique pouvant apparaître dans un problème de maths. Les algorithmes s'écrivent dans un langage de programmation qui,, pour beaucoup d'étudiants en classe préparatoire, est aussi Maple. La partie 2 de ce cours détaille les structures Maple servant surtout pour l'algorithmique.*

Ce poly a été fait avec la version 5 de Maple.

Partie 1 : les commandes de base

Partie 3 : compléments

Procédures (2)

Option *remember*

Accès aux paramètres

Structure d'une procédure

Partie 1 : les commandes de base

Quelques exemples de ce qu'on peut faire avec Maple

Le principe de Maple est très simple : on entre une ou plusieurs expressions, terminées par un point-virgule ou deux points selon que l'on désire ou non l'affichage de la réponse. Si l'on ne met rien Maple râle :

```
> 1+1; 1+1+1;
                                   2
                                   3
> 1+1: 1+1+1;
                                   3
> 1+1
>
Warning, premature end of input
```

La caractère appelé "ditto", à savoir % (à partir de la version 5 de Maple) ou " (jusqu'à la version 4), fait référence au dernier résultat obtenu :

```
> 1+1;
                                   2
> %+3;
                                   5
```

Les espaces entre les objets Maple sont facultatifs (sauf en cas d'ambiguïté). Un saut de ligne équivaut à un espace :

```
> 1+1; 1   +  1  ; 1+
  1;
                                   2
                                   2
                                   2
```

Voici ci-dessous quelques exemples d'expressions entrées dans Maple (version 5) et la réponse fournie par Maple.

```
> 2^500;
```

$$32733906078961418700131896968275991522166420460430647894832913680961337964046745548832700923259041571508866841275600710092172565458853930533285275893376$$

```
> 100!;
```

$$93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000$$

```
> 8/6;
```

$$\frac{4}{3}$$

```
> 1/3+3/5-1/7+5/9;
```

$$\frac{424}{315}$$

```
> a+a;
```

$$2\,a$$

```
> Pi;
```

$$\pi$$

```
> evalf(Pi);
```

$$3.141592654$$

```
> evalf(Pi,100);
```

$$3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706 8$$

```
> sqrt(2);
```

$$\sqrt{2}$$

```
> evalf(sqrt(2),50);
```

$$1.4142135623730950488016887242096980785696718753769$$

```
> sin(Pi/3);
```

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

```
> sum(k^2,k=1..10);
```

$$385$$

```
> sum(k^2,k=1..n);
```

$$\frac{1}{3}(n+1)^3-\frac{1}{2}(n+1)^2+\frac{1}{6}n+\frac{1}{6}$$

```
> factor(%);
```

$$\frac{1}{6}n\,(n+1)\,(2\,n+1)$$

```
> sum(x^k,k=0..n);
```

$$\frac{x^{(n+1)}}{x-1}-\frac{1}{x-1}$$

```
> simplify(%);
```

$$\frac{x^{(n+1)}-1}{x-1}$$

```
> sum(binomial(n,k),k=0..n);
```

$$2^n$$

```
> sum(k*binomial(n,k),k=0..n);
```

$$\frac{1}{2}\,2^{n}\,n$$

```
> simplify(%);
```

$$2^{(n-1)}\,n$$

```
> sum(x[k],k=0..5);
```

$$x_0 + x_1 + x_2 + x_3 + x_4 + x_5$$

```
> sum(x[k],k=1..n);
```

$$\sum_{k=1}^{n} x_k$$

```
> limit((x^2-1)/(x-1),x=2);
```

$$3$$

```
> limit(1/x,x=0);
```

$$\mathit{undefined}$$

```
> limit(1/x,x=0,left);
```

$$-\infty$$

```
> limit(sin(x)/x,x=0);
```

$$1$$

```
> limit(sin(a*x)/x,x=0);
```

$$a$$

```
> limit(sin(x),x=a);
```

$$\sin(a)$$

```
> limit(exp(x),x=-infinity);
```

$$0$$

```
> limit(exp(t*x),x=-infinity);
```

$$\lim_{x \to (-\infty)} \mathbf{e}^{(t\,x)}$$

```
> assume(t>0);
> %;
```

$$0$$

```
> limit((sin(x)-sin(a))/(x-a),x=a);
```

$$\cos(a)$$

```
> limit((f(x)-f(a))/(x-a),x=a);
```

$$\mathrm{D}(f)(a)$$

```
> diff(x^5+x^3,x);
```

$$5\,x^4 + 3\,x^2$$

```
> diff(f(sin(x)),x);
```

$$\mathrm{D}(f)(\sin(x))\,\cos(x)$$

```
> expand((a+b)^15);
```

$$a^{15} + 15\,b\,a^{14} + 105\,b^2\,a^{13} + 455\,b^3\,a^{12} + 1365\,b^4\,a^{11} + 3003\,b^5\,a^{10} + 5005\,b^6\,a^9 + 6435\,b^7\,a^8 + 6435\,b^8\,a^7 + 5005\,b^9\,a^6 + 3003\,b^{10}\,a^5 + 1365\,b^{11}\,a^4 + 455\,b^{12}\,a^3 + 105\,b^{13}\,a^2 + 15\,b^{14}\,a + b^{15}$$

```
> factor(x^2-4);
```

$$(x-2)\,(x+2)$$

```
> factor(x^2-(sqrt(2)+sqrt(3))*x+sqrt(6));
```

$$(x-\sqrt{2})\,(x-\sqrt{3})$$

```
> factor(x^2-2);
```

$$x^2 - 2$$

```
> factor(x^2-2,sqrt(2));
```

```
> factor(x^2+1);
```

$$(x+\sqrt{2})\,(x-\sqrt{2})$$

```
> factor(x^2+1,I);
```

$$x^2+1$$

$$(x+I)\,(x-I)$$

Expressions

Expressions composées

Exemple : On tape : `f(x+y,x*y);` Maple répond : $f(x+y, x\,y)$

Forme d'entrée : f(x+y,x*y);
Forme de sortie : $\mathbf{f(x+y, x\,y)}$ (le symbole de multiplication a disparu)
Forme interne : f (+ (x, y), * (x, y))

De façon générale, une expression composée e est de la forme $e_0(e_1, \ldots , e_n)$ où les e_i sont elles-mêmes des expressions. On les appelle les **opérandes** de l'expression e.

Expressions atomiques

Ce sont les expressions qui ne se décomposent pas en sous-expressions.

Nombres entiers : Exemples: 1984, -37

Symboles :
• Exemples : f, x, x1, x2, , truc, bidule, sin, ln, evalf, +, *, ^, =, :=
• Les accents graves permettent d'écrire des symboles contenant n'importe quel caractère. Par exemple, 1984 est un entier, `1984` est un symbole.
• S'il n'y a pas d'ambiguïté ou de problème de syntaxe les accents graves sont inutiles. Par exemple, les symboles `x` et x sont synonymes.
• Autre emploi des accents graves : on peut parfois s'en servir pour entrer une expression Maple directement sous sa forme interne. Par exemple, +(x,y) est refusé par l'analyseur syntaxique de Maple et renvoie un message d'erreur, mais `+`(x,y) est synonyme de x+y.

Chaînes de caractères : (à partir de la version 5). Exemple : "il fait beau"

Représentation des nombres non entiers

En pratique il peut être utile de considérer (même si ce n'est pas tout-à-fait vrai) les rationnels et les valeurs décimales comme des expressions composées. Exemples :

forme d'entrée	forme interne
8/3	fraction (8, 3)
34.57	float (3457, -2)

Représentation des 4 opérations

forme d'entrée	forme interne
a + b + c	+ (a, b, c)
a + (b + (c + d)))	+ (a, b, c, d, e)
- y	* (-1, y)
x - y	+ (x, * (-1, y))

x - 3 * y	+ (x, * (-3, y))
a * b * c	*(a, b, c)
a *(b * (c * d)))	* (a, b, c, d, e)
1 / y	^(y, -1)
x / y	* (x, ^(y, -1))
x / y^3	* (x, ^(y, -3))

Représentation des nombres complexes

Le complexe *i* est noté *I* et considéré par Maple comme (-1)^(1/2)

forme d'entrée	forme interne
I	^(-1, fraction (1, 2))
3 + 4 * I	+ (3, * (4, ^(-1, fraction (1, 2))))

Noms de variables

Un **nom** est soit un **symbole**, soit une **variable indexée**.
Une variable indexée s'écrit comme une expression composée, sauf que les parenthèses sont remplacées par des crochets. Les objets entre les crochets s'affichent comme des indices.

Exemple :

```
> x[i,j];
```

$$x_{i,j}$$

Evaluation des expressions

Evaluation d'une expression

Lorqu'on entre une expression, Maple **l'évalue**, et renvoie une nouvelle expression, qui est le **résultat** de la première.

En général le processus d'évaluation d'une expression composée est le suivant :
- exécution de simplifications automatiques (ex : + (a, + (b, c)) remplacé par + (a, b, c) , ou bien 1+1 remplacé par 2)
- évaluation des sous-expressions (opérandes)
- application d'une procédure (voir plus loin), dans le cas où la tête d'expression est le nom d'une procédure.

Blocage ou forçage d'évaluation

On peut bloquer l'évaluation d'une expression en l'enrobant d'apostrophes (accents aigus) :

```
> sin(Pi); 'sin(Pi)';
```

$$0$$

$$\sin(\pi)$$

Après évaluation de 'sin(Pi)' les apostrophes ont disparu mais l'évaluation du contenu est bloquée.

Si une expression est enrobée de plusieurs couches d'apostrophes, chaque évaluation supprime une couche et laisse l'intérieur intact :

```
> ''sin(Pi)'';%;%;
```

```
                                    'sin(π)'
                                    sin(π)
                                      0
```

Noter qu'il n'y a aucun moyen de bloquer les simplifications automatiques :

```
> cos((1+1)*Pi); 'cos((1+1)*Pi)';
                                      1
                                   cos(2 π)
```

Inversement, **eval** permet de forcer une évaluation bloquée :

```
> 'sin(Pi)'; eval('sin(Pi)');
                                    sin(π)
                                      0
```

Affectation

Une affectation est de la forme *x* := *y* (lire: *x* reçoit *y*)

x est n'importe quel nom (symbole ou variable indexée)

y est n'importe quelle expression.

Lorsque Maple évalue l'affectation *x* := *y*, d'une certaine manière il stocke la règle de transformation " remplacer *x* par *y* ". A chaque fois qu'il rencontrera le nom *x* lors d'une évaluation, il le remplacera par y.

Lorsque Maple évalue *x* := *y*, il commence par évaluer *y*, puis affecte le résultat au nom *x*. La règle de transformation stockée n'est donc pas à proprement parler " remplacer *x* par *y* " mais " remplacer *x* par le *résultat de l'évaluation* de *y* ".
En revanche le nom *x* n'est pas évalué. Toute nouvelle affectation d'un nom écrase l'affectation précédente.

Exemples:

```
> x := 3;
                                    x := 3
> x;
                                      3
> x := 1+1: x;
                                      2
```

Toute tentative d'affecter une valeur à un symbole prédéfini par Maple provoque un message d'erreur :

```
> sin := 1;
Error, attempting to assign to `sin` which is protected
```

Affectation auto-référente

Lors d'une affectation *x* := *y*, l'expression y peut contenir le nom *x* si celui-ci est affecté. Sinon il risque d'y avoir boucle sans fin et message d'erreur :

```
> x:=3; x;
```

$$x := 3$$

$$3$$

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

$$x := 4$$

$$4$$

```
> y:=y+1;
Warning, recursive definition of name
```

$$y := y + 1$$

```
> y;
Error, too many levels of recursion
```

Affectation retardée

Lorsque Maple évalue $x := y$, il affecte à x la *valeur immédiate* de y :

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

$$y := 1$$

$$x := 1$$

$$y := 2$$

$$1$$

Ce qui a été affecté à x, c'est la valeur de y au moment de *l'évaluation* de l'affectation $x := y$. Pour pouvoir se servir de la valeur de y au moment de *l'utilisation* de la règle $x := y$, il suffit d'affecter à x le symbole y en *retardant* l'évaluation de y :

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

$$y := 1$$

$$x := y$$

$$y := 2$$

$$2$$

Désaffectation

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

$$x := 1$$

```
> x;
```

$$1$$

```
> x:='x';
```

$$x := x$$

```
> x;
```

$$x$$

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

$$x := 1$$

$$1$$

$$x$$

Alias

Définir un alias consiste à donner un nom à une expression.

Contrairement à une affectation, un alias fonctionne aussi bien en entrée qu'en sortie.

Le seul alias prédéfini par Maple est I pour (-1)^(1/2).

Chaque fois qu'un alias est défini, Maple renvoie la séquence de tous les alias existants.

Exemples pour bien comprendre la différence entre affectation et alias :

```
> parmi := binomial; parmi(4,2); parmi(n,p);
```

$$parmi := \text{binomial}$$

$$6$$

$$\text{binomial}(n, p)$$

```
> restart; alias(parmi = binomial); parmi(4,2); parmi(n,p);
```

$$I, parmi$$

$$6$$

$$\text{parmi}(n, p)$$

Structures mathématiques usuelles

Listes, ensembles, séquences

• Une suite finie $(x_1, \ldots, x_n)$ se note $[x_1, \ldots, x_n]$ en Maple et s'appelle une **liste**.
• Un ensemble fini $\{x_1, \ldots, x_n\}$ se note $\{x_1, \ldots, x_n\}$ et s'appelle un **ensemble**. Comme en mathématiques, et contrairement au cas d'une liste, l'ordre des éléments n'intervient pas, et lorsqu'un même élément apparaît plusieurs fois il y a simplification automatique :

```
> [a,b,c,b,b]; {a,b,c,b,b}; {1+1, 1+1+1, 1+1};
```

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

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

$$\{3, 2\}$$

• Une **séquence** est une suite finie d'éléments "en vrac" séparés par des virgules:

```
> x:=a,b: x;
```

$$a, b$$

• La liste vide, l''ensemble vide, et la séquence vide se notent respectivement par les expressions [ ], { }, et le symbole NULL :

```
> [a,NULL]; a,b,NULL,c,d,e,NULL; [NULL];
```

$$[a]$$

$$a, b, c, d, e$$

$$[\,]$$

La séquence vide ne provoque aucun affichage :

```
> NULL;
```

• L'intervalle [a, b] se note a .. b. C'est une notation purement formelle utilisée pour délimiter des plages de valeurs de variables (voir plus loin).

Opérations sur les ensembles

```
> x:={1,2,3}; y:={3,4};
```

$$x := \{1, 3, 2\}$$

$$y := \{4, 3\}$$

```
> x union y; x union {4,5}; x intersect y; x minus y; x
  union z; x union {z};
```

$$\{1, 4, 3, 2\}$$
$$\{1, 5, 4, 3, 2\}$$
$$\{3\}$$
$$\{1, 2\}$$
$$z \text{ union } \{1, 3, 2\}$$
$$\{z, 1, 3, 2\}$$

Conversions

```
> x:=[a,b,c]; convert(x,set);
```

$$x := [a, b, c]$$
$$\{b, a, c\}$$

```
> x:={a,b,c}; convert(x,list);
```

$$x := \{b, a, c\}$$
$$[b, a, c]$$

Construction automatique de séquences

expr$*n* : répétition d'expression

```
> (1+1)$3;
```

$$2, 2, 2$$

Pour ceux qui doutent de l'intérêt de répéter la même chose plusieurs fois, voici un exemple d'utilisation :

```
> restart; e := exp(x^2); diff(e,x); diff(e,x,x);
  factor(diff(e,x$8));
```

$$e := \mathbf{e}^{(x^2)}$$
$$2\,x\,\mathbf{e}^{(x^2)}$$
$$2\,\mathbf{e}^{(x^2)} + 4\,x^2\,\mathbf{e}^{(x^2)}$$
$$16\,\mathbf{e}^{(x^2)}\,(105 + 840\,x^2 + 840\,x^4 + 224\,x^6 + 16\,x^8)$$

Devinette : pourquoi le *restart* ?

$*a* .. *b* : conversion d'intervalle en séquence

```
> [$1..10];
```

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

seq avec plage d'indices

```
> seq(i^2, i=1..10);
```

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

```
> [seq(ithprime(k),k=1..100)];
```

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541]

seq avec indices explicites

```
> x:=[3,5,13]; seq(i^2,i=x);
```

$$x := [3, 5, 13]$$

$$9, 25, 169$$

```
> seq(i,i="toto");
```

"t", "o", "t", "o"

Manipulation d'expressions

nops

```
> e:=f(x,y,z);
```

$$e := f(x, y, z)$$

```
> nops(e);
```

$$3$$

op

```
> op(2,e);
```

$$y$$

```
> op(0,e);
```

$$f$$

```
> op(1..2,e);
```

$$x, y$$

```
> op(-1,e);
```

$$z$$

```
> op(e);
```

$$x, y, z$$

```
> op([]);
> op(0,[a,b,c]), op(0,{a,b,c}); op(0,1..2); op(0,''x'');
```

list, *set*

..

uneval

Remarque : les entiers, symboles, chaînes de caractères, étant des expressions atomiques, n'ont pas d'opérandes. Cependant, op(0, ...) fournit dans ces cas une réponse qui indique le type de l'objet :

```
> op(0, 1984), op(0, toto), op(0,"toto");
```

integer, *symbol*, *string*

subsop

```
> subsop(1=a,e);
```

```
                                    f(a, y, z)
> subsop(0=g,e);
                                    g(x, y, z)
> subsop(1=a,0=g,e);
                                    g(a, y, z)
> subsop(1=a,subsop(0=g,e));
                                    g(a, y, z)
```

subs

```
> subs(x=a,e);
                                    f(a, y, z)
```

map

```
> map(f,[x,y,z]);
                                [f(x), f(y), f(z)]
> map(f,g(x,y,z));
                                g(f(x), f(y), f(z))
> l:=[seq(i*Pi/6,i=0..6)]; map(sin,l);
                    l := [0, 1/6 π, 1/3 π, 1/2 π, 2/3 π, 5/6 π, π]
                    [0, 1/2, 1/2 √3, 1, 1/2 √3, 1/2, 0]
```

Concaténation et suppression d'éléments

- **Pour concaténer des séquences :** les mettres en vrac, séparées par des virgules.

```
> s1:=a,b: s2:=c,d,e: s1,s2;
                                    a, b, c, d, e
```

- Pour concaténer des listes : concaténer les contenus et enrober de crochets.

```
> l1:=[a,b]: l2:=[c,d,e]: [op(l1),op(l2)];
                                  [a, b, c, d, e]
```

- Pour concaténer des ensembles : la concaténation s'appelle la réunion...

```
> e1:={a,b}: e2:={c,d,e}: e1 union e2;
                                  {d, e, a, b, c}
```

- Pour supprimer un élément d'une liste : le remplacer par NULL.

```
> l:=[a,b,c,d]: subsop(2=NULL,l);
                                     [a, c, d]
> l:=[a,x,b,x,c]: subs(x=NULL,l);
                                     [a, b, c]
```

- Pour supprimer des éléments d'un ensemble :

```
> e:={a,b,c,d}: e minus {a,b};
                                     {d, c}
```

- Pour concaténer des symboles ou des chaînes de caractères :
On peut utiliser *cat* ou mettre un point entre les objets à concaténer.

```
> cat(truc, bidule), truc.bidule;
                                   trucbidule
```

trucbidule

La différence est que le point bloque l'évaluation du premier objet à concaténer :

```
> truc := truc1: bidule := bidule1: cat(truc, bidule),
  truc.bidule;
```

truc1bidule1, *trucbidule1*

Le cas des chaînes de caractère est similaire. Une chaîne n'étant pas destinée à être évaluée, autant utiliser le point :

```
> "truc"."bidule";
```

"trucbidule"

Dans le cas d'objets de nature différente, c'est le premier qui détermine le type du résultat :

```
> truc."machin".bidule;
```

trucmachinbidule

A partir du deuxième objet à concaténer, on peut d'ailleurs mettre autre chose que des symboles ou des chaînes. Voici un exemple de construction automatique de séquence de symboles :

```
> seq(x.i, i=1..10);
```

x1, *x2*, *x3*, *x4*, *x5*, *x6*, *x7*, *x8*, *x9*, *x10*

qui peut d'ailleurs s'abréger en

```
> x.(1..10);
```

x1, *x2*, *x3*, *x4*, *x5*, *x6*, *x7*, *x8*, *x9*, *x10*

Exercices

1. Deviner le résultat de l'évaluation de chacune des expressions suivantes :

```
> op(2,I); op(0,op(2,I));
> op(0,8/3); op(0,3.141592);
> op(0,sin(Pi/6));
> op(0,'sin(Pi/6)');
> op(0,''sin(Pi/6)'');
```

2. Deviner le résultat de l'évaluation des expressions suivantes (bien réfléchir...):

```
> subs(-1=1,a-1/a); subs(x+y=t, x+y+z);
```

3. Deviner le résultat de l'évaluation de :

```
> x:=si: y:=n: eval(cat(x,y)(Pi));
```

Un exemple : calculs de sommes explicites

1. Somme des éléments d'une liste d'entiers :

```
> l:=[3,7,15,42,53];
```

$l := [3, 7, 15, 42, 53]$

```
> convert(l,`+`);
```

120

Autre solution :

```
> `+`(op(l));
```

120

2. Calcul de 1+2+...+10 :

```
> 1+2+3+4+5+6+7+8+9+10;
```

55

Problème : si on remplace 10 par 1000 ça devient chaud.

Mais un synomyme de l'expression précédente est

```
> add(i, i=1..10);
```

$$55$$

Et cette fois on peut aisément aller beaucoup plus loin que 10 :

```
> add(i, i=1..100000);
```

$$5000050000$$

Autres possibilités:

```
> `+`(seq(i,i=1..10));
```

$$55$$

```
> `+`($1..10);
```

$$55$$

```
> convert([$1..10],`+`);
```

$$55$$

3. *add* versus *sum*

Remplaçons *add* par *sum* :

```
> sum(i, i=1..10);
```

$$55$$

Cependant *add* et *sum* ne sont pas équivalents.
add(i, i=1..*n*) est remplacé par 1+2+...+*n* puis le calcul est effectué. Par conséquent, *n* est nécessairement un entier *effectif*.
En revanche *sum* sait traiter des sommes de longueur *symbolique* :

```
> sum(i,i=1..n);
```

$$\frac{1}{2}(n+1)^2-\frac{1}{2}n-\frac{1}{2}$$

```
> add(i,i=1..n);
Error, unable to execute add
```

4. Cas d'un produit

Remplacer respectivement *add* et *sum* par *mul* et *product*.

Relations et expressions booléennes

Symboles *true* et *false*

Les symboles *true* et *false* signifient respectivement, comme on s'en doute, *vrai* et *faux*.

Symboles de relation

Les symboles de relation sont les 6 symboles <, >, =, <>, <=, >=.
Une expression construite à l'aide de l'un de ces symboles représente une égalité ou une

inégalité. Une fois enrobée du symbole *evalb* ("évaluer en un booléen") l'expression représente une condition dont Maple va chercher à savoir si elle est vraie ou fausse. S'il est possible de décider de la réponse, le résultat de l'évaluation sera *true* ou *false*:

```
> 1<2; evalb(1<2);
```

$$1 < 2$$

$$\mathit{true}$$

```
> x:=1; x=1; evalb(x=1);
```

$$x := 1$$

$$1 = 1$$

$$\mathit{true}$$

Lorsqu'une inégalité fait intervenir des variables non affectées en général Maple ne peut pas conclure:

```
> evalb(y=y), evalb(y<y), evalb(y=z), evalb(y<z);
```

$$\mathit{true}, \mathit{false}, \mathit{false}, y - z < 0$$

Noter qu'evalb ne reconnaît une égalité comme vraie que si les deux membres sont syntaxiquement identiques.

assign

La commande *assign* permet de déclarer la ou les affectations correspondant à une égalité ou une liste ou un ensemble d'égalités.

```
> restart; x=1; x; assign(x=1); x;
```

$$x = 1$$

$$x$$

$$1$$

Connecteurs logiques

On peut combiner des expressions booléennes au moyen des connecteurs *and*, *or*, *not*. Dans le cas où une expression booléenne contient un connecteur, *evalb* est inutile et l'expression s'évalue d'elle-même :

```
> 1<2 and 3<2, 1<2 or 3<2, not(3<2);
```

$$\mathit{false}, \mathit{true}, \mathit{true}$$

is versus *evalb*

Essayons de comparer deux réels dont l'un au moins est symbolique :

```
> evalb(1<Pi);
```

$$1 - \pi < 0$$

Le problème est qu'*evalb* ne gère que l'aspect syntaxique : le symbole Pi est considéré comme n'ayant aucune signification, tout comme une variable.

La solution consiste à utiliser *is* à la place d' *evalb* :

```
> is(1<Pi);
```

$$\mathit{true}$$

select et *remove*

select permet de sélectionner les opérandes d'une expression vérifiant une certaine condition. Plus précisément, *select (f, e)* sélectionne les opérandes *x* de l'expression *e* tels que *f(x)* s'évalue en *true* :

```
> e:=[$1..10]; map(isprime, e); select(isprime, e);
```

$$e := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

$$[false, true, true, false, true, false, true, false, false, false]$$

$$[2, 3, 5, 7]$$

Inversement, *remove* supprime les opérandes vérifiant la condition :

```
> remove(isprime, e);
```

$$[1, 4, 6, 8, 9, 10]$$

Fonctions

Ecriture d'une fonction

Exemple :

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

$$f := x \rightarrow x^2 + x + 1$$

$$13$$

```
> (x->x^2+x+1)(3);
```

$$13$$

On peut aussi écrire des fonctions de plusieurs variables :

```
> f := (x,y)->x+y; f(3,4);
```

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

$$7$$

Noter qu'il y a finalement deux façons de calculer la valeur d'une fonction en un point :
- soit on déclare une *expression* et on utilise *subs* :

```
> e := x^2+x+1: subs(x=3,e);
```

$$13$$

- soit on déclare une *fonction* et on l'*applique* :

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

$$13$$

Evaluation d'une fonction

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

$$f$$

$$x \rightarrow x^2 + x + 1$$

Moralité: le nom d'une fonction ne s'évalue pas. Il faut forcer l'évaluation avec *eval*.

Valeurs particulières

On peut déclarer un nombre fini de valeurs particulières pour une fonction. Exemples :

```
> f:=x->cos(x)/(1+x): f(0):=1000:
  f(Pi), f(Pi/2), f(0);
```

$$-\frac{1}{1+\pi}, 0, 1000$$

Les valeurs particulières se déclarent *après* la forme générale.

Récursivité

La récursivité est le fait qu'une fonction puisse s'appeler elle-même.

Exemple : on veut écrire une fonction *fact* telle que fact(*n*) renvoie la valeur de *n*! si n est un entier positif.

- Ecriture correspondant à *n*! = produit des entiers de 1 à *n* :

```
> fact := n -> product(k,k=1..n); fact(5);
```

$$fact := n \rightarrow \prod_{k=1}^{n} k$$

$$120$$

- L'écriture récursive correspnd à la définition mathématique suivante : *n*! est le terme d'indice *n* de la suite récurrente (u_n) définie par $u_0=1$, $u_n=n*u_{n-1}$.

```
> fact := n -> n*fact(n-1); fact(0) := 1; fact(5);
```

$$fact := n \rightarrow n\ \mathrm{fact}(n-1)$$

$$\mathrm{fact}(0) := 1$$

$$120$$

unapply

Il faut savoir que lors de la déclaration d'une fonction, la partie droite de la flèche n'est pas évaluée, ce qui peut conduire à quelques désagréments.

Exemple :

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

$$x^2 + x + 1$$

Explication : dans *f* := *x* -> *e*, *e* n'est pas évalué. *f*(3) s'évalue donc en *e*, qui ensuite s'évalue en $x^2 + x + 1$.

Pour y remédier : remplacer *x* -> *e* par unapply(*e*, *x*) :

```
> f:=unapply(e,x): f(3);
```

$$13$$

Pourquoi "unapply"? Quand on applique une fonction *f* à *x* on obtient l'expression *f(x)*; et donc, inversement, on "désapplique" l'expression par rapport à la variable *x* pour retouver la fonction.

Types

Le type de base d'une expression

Toute expression possède un unique *type de base*, qui indique la nature de l'objet mathématique qu'elle représente.

Le type de base d'une expression est donné par la fonction *whattype*.
Très souvent, le type de base d'une expression est son opérande 0, de sorte que *whattype*(...) équivant à *op*(0, ...)

```
> whattype(8/3);
```

fraction

```
> map(whattype, [3.14159, 1984, Pi, "coucou",x+y,x<y,x=y,a and b, {x,y},[x,y]]);
```

[*float*, *integer*, *symbol*, *string*, +, <, =, *and*, *set*, *list*]

```
> s:= a,b,c: whattype(s);
```

exprseq

```
> whattype(''a'');
```

uneval

```
> whattype(x[i]);
```

indexed

Attention, le type *function* n'est pas le type des fonctions, mais des expressions de type inconnu (dont l'opérande 0 n'est pas un type de base) :

```
> whattype(f(x,y,z));
```

function

Les fonctions, elles, ont le type *procedure*. L'explication sera donnée plus loin. De plus, ne pas oublier que le nom d'une fonction ne s'évalue pas de lui-même :

```
> f := x->x^2: f, whattype(f), eval(f), whattype(eval(f));
```

f, *symbol*, $x \rightarrow x^2$, *procedure*

Autres types simples

Comme on peut le constater sur les exemples précédents, le nom d'un type de base est un symbole. Un tel type s'appelle un type *simple*.
Il se trouve qu'il y a d'autres types simples que les types de base. La raison est la suivante : une expression peut être, du point de vue mathématique, de plusieurs types. Par exemple, tout entier est un rationnel.
Mais si l'on veut savoir si 1984 est un rationnel, on ne peut le savoir par *whattype*, puisqu'on aura seulement la réponse *integer*.
Le type simple *rational* couvre les rationnels mathématiques, c'est-à-dire justement les expressions de type de base *integer* ou *fraction*.
On ne peut pas y accéder par *whattype*, mais par *type*, qui permet de tester si une expression est d'un type donné :

```
> type(1984, integer); type(1984, fraction); type(1984, rational);
```

true
false
true

En cherchant *type* dans l'aide vous pouvez tout savoir sur les divers types simples.

Voici quelques types simples très utiles en pratique :

numeric (réel effectif: *integer* ou *fraction* ou *float*)
realcons : réel (effectif ou symbolique)
complex
name : *symbol* ou *indexed* (tout ce qui peut subir une affectation)
anything : comme son nom l'indique...
logical : <, >, =, etc

relation : *and*, *or*, *not*
boolean : *logical* ou *relation* ou *true* ou *false*
positive, *negative*, *nonneg*, *posint*, *negint*, *nonnegint*
algebraic : tout ce qui peut désigner un nombre (par exemple un nom, mais pas une liste, un ensemble, une égalité...)

Types structurés

Les types structurés sont les types dont le nom n'est pas un symbole mais une expression composée.

Exemples:

```
> type(3+4*I, complex(integer));
```
true

```
> type([1,x,y], [integer, name, symbol]);
```
true

```
> type([1,2,3], list(integer));
```
true

```
> type({1,2,3}, set(integer));
```
true

```
> type(x, {name, integer});
```
true

```
> type(x=1, name=integer);
```
true

assume, additionally, about, is

On peut faire des hypothèses sur un symbole à l'aide de *assume*. Dans ce cas la variable s'affiche suivie d'un tilde.
On peut ensuite rajouter des hypothèses sur ce même nom à l'aide de *additionally* (un nouvel *assume* écrase le précédent)
On peut avoir accès aux hypothèses faites sur un nom à l'aide de *about*.

Exemples:

```
> cos(n*Pi);
```
$\cos(n\,\pi)$

```
> assume(n,integer);
> cos(n*Pi);
```
$(-1)^{n\sim}$

```
> about(n);
Originally n, renamed n~:
  is assumed to be: integer
> additionally(n>0);
> about(n);
Originally n, renamed n~:
  is assumed to be: AndProp(RealRange(1,infinity),integer)
> sqrt(x^2);
```
$\sqrt{x^2}$

```
> assume(x>0);
> sqrt(x^2);
```
$x\sim$

La commande *type* ne manipule que des types syntaxiques. Elle ne connaît donc pas les

hypothèses faites avec *assume* :

```
> assume(p,integer); type(p, integer);
                                    false
```

Il suffit d'utiliser *is* :

```
> is(p, integer);
                                    true
```

Partie 2 : les structures utiles en algorithmique

Tests et boucles

Tests

Un test est une structure du type *si ... alors ... sinon ...* Il s'écrit en Maple *if ... then ... else ... fi*. Le *fi* final est obligatoire et sert à terminer l'instruction . La condition s'évalue d'elle-même sans qu'il soit nécessaire de l'enrober d'un *evalb*.

```
> if 0=1 then x else y fi;
                                    y
```

Remarque : chacun des deux cas du test peut être constitué de plusieurs expressions. Dans ce cas, fait que les séparateurs soient des points-virgules ou deux-points n'intervient pas. C'est le signe venant après le *fi* qui détermine l'affichage :

```
> if 0=0 then a; b: c else d fi;
                                    a
                                    b
                                    c
> if 0=0 then a; b: c else d fi:
```

Si la condition du test ne s'évalue pas en *true* ou *false*, Maple renvoie un message d'erreur :

```
> if a<b then x else y fi;
Error, cannot evaluate boolean
```

Si le résultat d'un test doit être imbriqué dans une plus grosse expression, le test doit être mis sous forme interne :

```
> a:=3: b:=4: Pi + (if a<b then Pi else 0 fi);
reserved word `if` unexpected
> a:=3: b:=4: Pi + `if`(a<b, Pi,0);
                                    2 π
```

Bien évidemment on peut aussi s'arranger pour ne pas avoir à imbriquer le test :

```
> x := (if 0=0 then 1 else 2 fi): x;
reserved word `if` unexpected
> if 0=0 then x:=1 else x:=2 fi: x;
                                    1
```

On peut évidemment imbriquer des tests les uns dans les autres : dans *if a then b else c fi*, les objets *b* et *c* peuvent eux-mêmes être des tests.
La forme *if a then b else if c then d else e fi fi*
s'abrège en *if a then b elif c then d else e fi*.
On peut intercaler autant de *elif* que l'on souhaite, et l'on termine par un seul *fi*.

Boucles *for*

Il y a deux types de structures appelées "boucle for" en Maple :

- Une structure du type "pour *i* (l'indice de la boucle) variant de telle valeur (la valeur de départ) à telle autre (la valeur d'arrivée) , en augmentant de telle quantité (le pas) à chaque fois, faire telle chose" :

```
> for i from 1 to 10 by 2 do i^2 od;
```

1
9
25
49
81

- Une structure du type "pour *i* prenant telle suite finie de valeurs faire telle chose" :

```
> l := [7,10,23,41]: for i in l do i^2 od;
```

49
100
529
1681

Remarques :
- Comme dans le cas des tests, pour toutes les boucles on peut mettre plusieurs instructions entre le *do* et le *while*, mais c'est le signe de ponctuation venant après le *od* qui détermine l'affichage.
- L'indice *i* est une variable muette : on peut évidemment le remplacer par n'importe quel symbole.
- Dans la première forme les champs *from* et *by* sont facultatifs. La valeur par défaut est 1.
- Dans la première forme, les valeurs de départ et d'arrivée et le pas ne sont pas forcément entiers. La boucle s'arrête dès que la valeur d'arrivée est dépassée.
- Dans la première forme, le pas peut être négatif (dans ce cas la suite des valeurs successives de i est décroissante).
- Dans la seconde forme, l'indice *i* ne parcourt pas forcément une liste : cela peut être n'importe quelle expression. Les valeurs de *i* sont les opérandes de cette expression :

```
> for i in x+y+z do i^2 od;
```

x^2
y^2
z^2

Boucles *while*

La "boucle while" est une structure du type "faire telle chose tant que telle condition est vérifiée".
On peut d'ailleurs simuler une boucle *for* au moyen d'une boucle *while* :

```
> i:=1: while i<=10 do print(i^2); i:=i+2 od:
```

1
9
25
49
81

Devinette : pourquoi ai-je bloqué l'affichage avec les deux points et forcé un affichage avec *print* ?

Réponse :

```
> i:=1: while i<=10 do i^2; i:=i+2 od;
```

1

$i := 3$

9

$i := 5$

25

$i := 7$

49

$i := 9$

81

$i := 11$

Boucles générales

On paut mixer chacune des deux boucles *for* avec un *while*, ce qui donne les structures suivantes :
for i from a to b by c while d do ... od;
for i in a while b do ... od;

De l'utilité des boucles en Maple

L'usage des boucles est très souvent inutile en Maple.

- Exemple 1 : reprenons le cas de l'affichage des carrés des entiers de 1 à 10 par pas de 2. Au lieu de procéder comme précédemment, autant faire :

```
> seq((2*i+1)^2, i=0..4);
```

1, 9, 25, 49, 81

- Exemple 2 : calcul de la somme des entiers de 1 à 100.
On peut écrire une boucle (ne pas oublier les deux points):

```
> s:=0: for i to 100 do s:=s+i od: s;
```

5050

Mais autant faire:

```
> add(i,i=1..100);
```

5050

Cependant, prendre garde aux règles du jeu des concours:

- Maple pour faire des maths : Là, il s'agit d'utiliser au mieux la puissance de Maple pour arriver rapidement au résultat : éviter les boucles au maximum.

- Maple pour écrire des algorithmes : Là, vous n'aurez droit qu'aux structures qui existent dans tous les langages de programmation. Vous ne pouvez donc pas utiliser les structures puissantes spécifiques à Maple comme les séquences, listes, ensembles. Vous devez donc écrire des boucles. Voir plus loin pour la liste complète des instructions autorisées pour l'algorithmique.

Tables et tableaux

Tables

Supposons que l'on souhaite manipuler un ensemble de données de façon à pouvoir accéder à l'une des données, ou supprimer ou rajouter une donnée.

On peut évidemment utiliser une liste :
- on accède au *ième* élément d'une liste *l* par *op(i, l)* ;
- on supprime le *ième* élément de *l* par *l := subsop(i=NULL, l)* ;
- on rajoute un élément *x* à la fin de la liste *l* par *l* := [*op(l)*, *x*] ;

Quel est le problème? Toutes ces instructions nécessitent de parcourir une partie de la liste et prennent, dans le pire cas, un temps proportionnel à la longueur de la liste et non pas un temps constant.

Or, en informatique, on manipule fréquemment des ensembles de données gigantesques (penser par exemple à l'annuaire du téléphone), et le temps de traitement d'une donnée ne doit pas dépendre (ou très peu) du nombre de données.

La solution en Maple consiste à utiliser des affectations en vrac de variables indexées :

```
> jour[1]:=lundi: jour[2]:=mardi: jour[3]:= mercredi:
  jour[4]:= jeudi:
  jour [5]:=vendredi: jour[6]:=samedi:
```

On modifie ou on rajoute un élément par une nouvelle affectation :

```
> jour[7]:=dimanche;
```

$$jour_7 := dimanche$$

Pour accéder à un élément :

```
> jour[1]; jour[8];
```

$$lundi$$
$$jour_8$$

Qu'est-ce donc qu'une table? C'est tout simplement un ensemble de données, chaque donnée étant affectée à une variable indexée, les noms de ces variables étant constitués d'un même symbole (*jour* dans l'exemple précédent) et d'un (ou plusieurs) indice.

Apparemment, ce que l'on perd, par rapport aux listes, c'est la possibilité de récupérer d'un seul coup toutes les données. Mais c'est encore possible : le nom de la table (le symbole) s'évalue en lui-même, mais en l'enrobant d'un *eval* on récupère une table, c'est-à-dire un objet contenant une liste d'égalités :

```
> jour; eval(jour);
```

$$jour$$

$$\begin{aligned}&table([\\&\quad 1 = lundi\\&\quad 2 = mardi\\&\quad 3 = mercredi\\&\quad 4 = jeudi\\&\quad 5 = vendredi\\&\quad 6 = samedi\\&\quad 7 = dimanche\\&\quad])\end{aligned}$$

Les indices ne sont pas forcément des entiers, et peuvent même être des séquences d'objets :

```
> a[x]:=1: a[y,z]:=2: eval(a);
table([
```

```
    (y, z) = 2
    x = 1
    ])
```

Copie d'une table

```
> Jour:=jour; Jour[1];
```

$$Jour := jour$$

$$lundi$$

Jusque là rien que de très normal : *Jour* s'évalue en *jour* et donc *Jour*[1] en *jour*[1] c'est-à-dire *lundi*.

```
> jour[1]:=truc: jour[1]; Jour[1];
```

$$truc$$

$$truc$$

Là encore, c'est normal.
Si l'on veut que Jour[1] conserve l'ancienne valeur *lundi*, il aurait fallu faire, au lieu de *Jour* :=*jour*, une copie séparée de *jour*[1] par *Jour*[1]:=*jour*[1] :

```
> Jour1[1]:=jour[1]: jour[1]:=bidule: jour[1]; Jour1[1];
```

$$bidule$$

$$truc$$

Si l'on veut faire une copie d'une table qui en soit indépendante (c'est-à-dire que toute modification ultérieure de la table sera sans influence sur la copie) il faut donc refaire une par une toutes les affectations. Mais on peut aussi utiliser copy :

```
> jour[1]:=lundi: jour[2]:=mardi: jour[3]:= mercredi:
  jour[4]:= jeudi:
  jour [5]:=vendredi: jour[6]:=samedi: jour[7]:=dimanche:
  Jour:=jour: Jour1:=copy(jour): eval(Jour); eval(Jour1);
table([
    1 = lundi
    2 = mardi
    3 = mercredi
    4 = jeudi
    5 = vendredi
    6 = samedi
    7 = dimanche
    ])
table([
    1 = lundi
    2 = mardi
    3 = mercredi
    4 = jeudi
    5 = vendredi
    6 = samedi
    7 = dimanche
```

```
    ])
> jour[1]:=truc: eval(Jour); eval(Jour1);
table([
    1 = truc
    2 = mardi
    3 = mercredi
    4 = jeudi
    5 = vendredi
    6 = samedi
    7 = dimanche
    ])
table([
    1 = lundi
    2 = mardi
    3 = mercredi
    4 = jeudi
    5 = vendredi
    6 = samedi
    7 = dimanche
    ])
```

Tableaux

Un tableau est une structure décrivant une table à un ou plusieurs indices, dont chaque indice est un entier assujetti à parcourir un certain intervalle.
Comme pour une table le nombre d'indices est quelconque. Voici un exemple de tableau à deux indices :

```
> a:=array(1..2,1..3); eval(a); print(a);
```

$$a := \text{array}(1 .. 2, 1 .. 3, [\ ])$$

$$\begin{bmatrix} ?_{1,1} & ?_{1,2} & ?_{1,3} \\ ?_{2,1} & ?_{2,2} & ?_{2,3} \end{bmatrix}$$

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix}$$

Si l'on sort de la plage d'indices on obtient un message d'erreur :

```
> a[1,4];
Error, 2nd index, 4, larger than upper array bound 3
```

On peut ensuite remplir le tableau :

```
> a[1,1]:=x: print(a);
```

$$\begin{bmatrix} x & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix}$$

On peut remplir un tableau d'un seul coup lors de sa construction :

```
> a:=array(1..2,1..3,[[1,2,3],[4,5,6]]); eval(a); a[2,3];
```

$$a := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$6$$

On peut accéder à toutes les caractéristiques du tableau :

```
> eval(a); op(0,eval(a)); op(2,eval(a)); op(3,eval(a));
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$array$$

$$1 .. 2, 1 .. 3$$

$$[(2, 2) = 5, (1, 2) = 2, (2, 1) = 4, (1, 1) = 1, (2, 3) = 6, (1, 3) = 3]$$

Procédures (1)

Ecriture d'une procédure

Les fonctions (écrites avec une flèche) sont des cas particuliers de procédures.

Par exemple, au lieu d'écrire

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

$$f := x \rightarrow x^2 + x + 1$$

$$13$$

on pouvait écrire :

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

$$f := \mathbf{proc}(x)\ x\text{^}2 + x + 1\ \mathbf{end}$$

$$13$$

Pourquoi écrire une procédure plutôt qu'une fonction? Considérons l'exemple suivant : on veut écrire la fonction qui à l'entier n associe la valeur de la somme 1+...+n, calculée par Maple.

On peut évidemment écrire (voir la section duivante pour la signification du message d'alerte):

```
> f:=n->add(i,i=1..n):
Warning, `i` in call to `add` is not local
> f(4);
```

$$10$$

Supposons que l'on veuille que le calcul de la somme se fasse à l'aide d'une boucle. On ne peut pas écrire :

```
> f:=n->(s:=0: for i from 1 to n do s:=s+i od: s):
`:=` unexpected
```

car la partie droite d'une flèche ne peut pas contenir d'affectation.

Mais on peut écrire :

```
> f:=proc(n) s:=0: for i from 1 to n do s:=s+i od: s end:
Warning, `s` is implicitly declared local
Warning, `i` is implicitly declared local
```

```
> f(4);
```
10

La variable n s'appelle le "paramètre" ou l'"argument" de la procédure.

De façon générale,
- lorsque l'on fait des maths avec Maple, on écrit des fonctions mathématiques (que l'on veut dériver, intégrer, dont on veut afficher la courbe, etc...) qui se calculent par des expressions simples : on utilise plutôt la flèche.
- lorsque l'on fait de l'algorithmique avec Maple, on écrit des algorithmes utilisant des affectations et des tests et/ou des boucles : on utilise des procédures.

Un procédure peut évidemment avoir plusieurs paramètres :
```
> somme := proc(x,y) x+y end: somme(3,4);
```
7

Comme dans le cas d'une fonction, on peut définir pour une procédure un nombre fini de valeurs particulières, déclarées *après* la procédure.

Variables locales et variables globales

Dans l'exemple précédent, le calcul de $f(n)$ utilise une variable auxiliaire s servant à stocker les valeurs successives de la somme dans la boucle. Supposons que, lors de l'appel $f(4)$, on ait déjà une variable s en cours d'utilisation, affectée précédemment durant la session Maple. Lors de l'évaluation de s:=0, la valeur précédente de s risque d'être écrasée, et donc perdue.

La solution consiste à déclarer que la variable s figurant dans la procédure est une variable *locale* à la procédure (les messages d'alerte affichés ci-dessus montrent que Maple, en l'absence d'information, a décidé de lui-même de considérer s comme une variable locale, mais ce n'est pas une bonne façon de procéder : il vaut mieux prévoir et contrôler ce que l'on fait).

Cela signifie qu'au début de l'exécution de la procédure, et donc avant d'affecter la valeur 0 à s, Maple prend soin de sauvegarder la valeur d'une éventuelle variable s en cours d'utilisation. Cette valeur est ensuite automatiquement rétablie une fois l'exécution de la procédure terminée.

Dans l'exemple précédent, la variable i servant d'indice dans la boucle doit elle aussi être "localisée".

La bonne façon d'écrire la procédure précédente est la suivante :

```
> f:=proc(n)
  local s,i:
  s:=0: for i from 1 to n do s:=s+i od: s
  end:
```

Le paramètre de la procédure (ici la variable n) est, lui, *automatiquement* localisé.

Inversement, si l'on veut que la variable s soit la même que celle qui existe "en dehors" de la procédure, il faut la déclarer *globale*. Les variables globales se déclarent *après* les variables locales :
proc(...) *local* ... : *global* ...: ... *(corps de la procédure)* ... *end*;

Un cas typique d'utilisation d'une variable globale est le suivant : si l'on veut que le contenu d'une variable utilisée par une procédure reste accessible une fois l'exécution de la procédure terminée, la variable doit être globale (une variable locale a une "durée de vie" qui est celle de l'exécution de la procédure).

Remarque : à l'intérieur des procédures j'ai systématiquement mis deux points, mais c'est sans importance : lors de l'exécution d'une procédure, la seule chose qui s'affiche est le résulat de l'appel de la procédure, qui est la résultat de la dernière expression évaluée dans la procédure.

Sauf, évidemment, si l'on force des affichages intermédiaires avec *print :*

```
> f:=proc(n) toto; print(titi); tata; tutu; n^2 end; f(3);
```

f := **proc**(*n*) *toto*; print(*titi*); *tata*; *tutu*; *n*^2 **end**

titi

9

Récursivité

Tout comme une fonction, une procédure peut s'appeler elle-même.
Exemple :

```
> fact:=proc(n) n*fact(n-1) end; fact(0):=1;
```

fact := **proc**(*n*) *n*∗fact(*n* – 1) **end**

fact(0) := 1

```
> fact(5);
```

120

C'est tout un art d'apprendre à "penser récursif" dans l'écriture d'une procédure.
Exemple : on veut écrire une procédure qui, lorsqu'elle reçoit en entrée une liste *l* , renvoie la même liste écrite "à l'envers".

1ère méthode : utilisation de *seq*

```
> renverser := proc(l)
  local n,i:
   n:=nops(l):
   [seq(op(-i,l),i=1..n)]
  end;
```

renverser := **proc**(*l*) **local** *n*, *i*; *n* := nops(*l*); [seq(op(–*i*, *l*), *i* = 1 .. *n*)] **end**

```
> renverser([a,b,c,d]); renverser([]);
```

[*d*, *c*, *b*, *a*]

[ ]

2ème méthode : méthode itérative (boucle)

```
> renverser := proc(l)
  local n,s,i:
   n:=nops(l): s:=NULL:
   for i from 1 to n do s:=s,op(-i,l) od:
   [s]
  end:
> renverser([a,b,c,d]); renverser([]);
```

[*d*, *c*, *b*, *a*]

[ ]

3ème méthode : méthode récursive

```
> renverser := proc(l)
  [op(-1,l),op(renverser([op(1..nops(l)-1, l)]))]
  end:
  renverser([]):=[]:
> renverser([a,b,c,d]); renverser([]);
```

[*d*, *c*, *b*, *a*]
[ ]

Les objets Maple utilisables en algorithmique

Dans une épreuve d'algorithmique aux concours on évalue l'aptitude du candidat à écrire des algorithmes en utilisant des tests et des boucles. N'est donc autorisé qu'un sous-ensemble de Maple contenant des structures qui existent dans la plupart les langages de programmation courants.

Voici plus précisément les objets Maple utilisables en algorithmique:

- les symboles pour pouvoir écrire des noms de variables.
- l'affectation :=.
- les objets de type booléen; les constantes *true* et *false*; les opérateurs *or*, *and*, *not*.
- les entiers; les opérations +, -, *, ^, *iquo*, *irem*, *mod* sur les entiers.
- les flottants; les opérations +, -, *, /, ^ sur les flottants; *floor* (partie entière), *evalf*.
- les symboles de relation =, <, >, <=, >=, <> pour comparer des nombres.
- les tests.
- les boucles.
- les procédures; les variables locales et globales; la récursivité.
- les tableaux; les variables indexées pour repéer les éléments d'un tableau.
- les chaînes de caractères; l'opérateur de concaténation *cat*.
- l'instruction *print*.

Tout le reste est interdit, en particulier toutes les structures propres à Maple : listes, ensembles, séquences.

- éviter la déclaration de valeurs particulières en dehors d'une procédure (spécificité propre à Maple) : faire plutôt un test à l'intérieur de la procédure. Par exemple, une procédure récursive du calcul de la fonction factorielle peut s'écrire

```
> fact := proc(n) n*fact(n-1) end: fact(0):=1:
```

mais on peut aussi écrire :

```
> fact := proc(n) if n>0 then n*fact(n-1) else 1 fi end:
```

Partie 3 : compléments

Procédures (2)

Cette section est constituée de commandes un peu plus "pointues" et pourra être sautée dans un premier temps. De toute façon, les commandes présentées ici sont spéciales à Maple et ne peuvent pas être utilisées en algorithmique.

Option *remember*

On peut déclarer des *options* à l'intérieur d'une procédure, de sorte que la forme générale d'une procédure est
proc(...) *local* ... : *global* ...: *option* ...: ... (*corps de la procédure*) ... *end*;

La seule option intéressante en pratique est l'option *remember*.

Cette option concerne les valeurs particulières. Maple stocke ces valeurs dans une table attachée à la procédure. Lorsque l'option *remember* est activée, Maple, après avoir exécuté la procédure, stocke le résultat dans la table des valeurs particulières, de sorte que, lors d'un nouvel appel de la procédure sur le même paramètre, il ne calculera plus rien mais ira lire le résultat dans la table.

Si l'on fait par exemple

```
> f:=proc(x) option remember: x^2+x+1 end: f(3); f(3);
                                   13
                                   13
```

le premier résultat f(3)=13 est calculé alors que le second est lu dans la table.

La table des valeurs particulières d'une procédure porte d'ailleurs le nom de *remember table*.

Dans une procédure récursive, l'option remember entraîne donc le strockage des résultats de tous les appels récursifs. Exemple :

```
> fact:=proc(n) option remember: n*fact(n-1) end;
  fact(0):=1; fact(5);
```

fact := **proc**(*n*) **option** *remember*; *n*∗fact(*n* − 1) **end**

fact(0) := 1

120

Maintenant, Maple "connaît" *fact*(0), *fact*(1), *fact*(2), *fact*(3), *fact*(4), *fact*(5); si l'on fait ensuite

```
> fact(7);
                                  5040
```

seuls les appels *fact*(7) et *fact*(6) provoquent l'exécution de la procédure *fact*.

L'option remember est surtout utile dans un procédure récursive comportant au moins *deux* appels récursifs.
Exemple : calcul des nombres de Fibonacci.
La suite de Fibonacci est la suite d'entiers (F_n) définie par récurrence par $F_0=0$, $F_1=1$, $F_{n+2} = F_{n+1} + F_n$.

Si l'on fait

```
> fibo := proc(n) fibo(n-1)+fibo(n-2) end: fibo(0):=0:
  fibo(1):=1: fibo(5);
                                   5
```

l'appel *fibo*(5) provoque les appels *fibo*(4) et *fibo*(3), l'appel *fibo*(4) provoque lui-même les appels *fibo*(3) et *fibo*(2), etc..., et la même chose est calculée plusieurs fois. On peut d'ailleurs montrer que temps de calcul de l'appel *fibo*(*n*) est exponentiel en *n*.

En revanche si l'on écrit

```
> fibo := proc(n) option remember: fibo(n-1)+fibo(n-2) end:
  fibo(0):=0: fibo(1):=1: fibo(5);
                                   5
```

chaque valeur calculée par *fibo* est stockée dans la table des valeurs particulières et le temps de calcul devient linéaire.

Accès aux paramètres

A l'intérieur d'une procédure on peut utiliser les variables prédéfinies suivantes :
args : la séquence des paramètres ("arguments") ayant été transmis à la procédure.
nargs : le nombre d'arguments.
procname : le nom de la procédure.

Une procédure peut d'ailleurs avoir un nombre d'arguments non spécifié. Dans ce cas on écrit *proc*().
Exemple :

```
> somme := proc() `+`(args) end; somme(4,7,10);
```

$$somme := \mathbf{proc}()\ `+`(\mathrm{args})\ \mathbf{end}$$

$$21$$

En guise d'exemple récapitulatif, essayons d'écrire une procédure qui calcule la valeur absolue d'un réel (chose idiote en pratique car Maple sait le faire tout seul, mais c'est un exercice d'entraînement). En général on écrit une procédure en supposant qu'elle sera correctement utilisée, mais ici, pour corser le problème, on veut gérer soi-même tous les cas possibles de mauvaise utilisation.

1er essai :

```
> valeurabs:=proc(x) if x>0 then x else -x fi end:
```

Problème :

```
> valeurabs(3); valeurabs(-5); valeurabs(Pi);
```

$$3$$

$$5$$

```
Error, (in valeurabs) cannot evaluate boolean
```

2ème essai :

```
> valeurabs:=proc(x) if is(x>0) then x else -x fi end:
```

Problème :

```
> valeurabs(-Pi); valeurabs(a);
```

$$\pi$$

$$-a$$

3ème essai :

```
> valeurabs:=proc(x)
    if type(x,realcons)
    then if is(x>0) then x else -x fi
    else 'valeurabs'(x)
    fi
  end:
```

Problème :

```
> valeurabs(Pi); valeurabs(a); valeurabs(a=b);
```

$$\pi$$

$$\mathrm{valeurabs}(a)$$

$$\mathrm{valeurabs}(a = b)$$

4ème essai :

```
> valeurabs:=proc(x)
    if type(x, algebraic)
    then
     if type(x,realcons)
     then if is(x>0) then x else -x fi
     else 'valeurabs'(x)
     fi:
    else ERROR(`l'argument doit représenter un nombre`)
    fi
  end:
```

Problème :

```
> valeurabs(a=b); valeurabs(3,5);
Error, (in valeurabs) l'argument doit représenter un nombre
```

$$3$$

5ème essai :

```
> valeurabs:=proc(x)
   if nargs<>1
   then ERROR(`la procédure `.procname.` a été appelée avec
  `.nargs.` arguments au lieu d'un seul `)
   elif type(x, algebraic)
   then
    if type(x,realcons)
    then if is(x>0) then x else -x fi
    else 'valeurabs'(x)
    fi:
   else ERROR(`l'argument doit représenter un nombre`)
   fi
  end:
> valeurabs(3,5);
Error, (in valeurabs) la procédure valeurabs a été appelée avec 2
arguments au lieu d'un seul
```

Cette fois c'est bon :

```
> valeurabs(-3); valeurabs(-sqrt(2)); valeurabs(x+y);
  valeurabs(x<y); valeurabs(3,5);
```

$$3$$

$$\sqrt{2}$$

$$\text{valeurabs}(x + y)$$

```
Error, (in valeurabs) l'argument doit représenter un nombre
Error, (in valeurabs) la procédure valeurabs a été appelée avec 2
arguments au lieu d'un seul
```

Structure d'une procédure

Comme pour une fonction, le nom d'une procédure ne s'évalue qu'avec *eval* :

```
> valeurabs; eval(valeurabs);
```

valeurabs

proc(*x*)
  if nargs ≠ 1 **then** ERROR(`*la procédure* `.procname.` *a été appelée avec* `.nargs.
    ` *arguments au lieu d'un seul* `)
  elif type(*x*, *algebraic*) **then**
    if type(*x*, *realcons*) **then if** is(0 < *x*) **then** *x* **else** –*x* **fi else** '*valeurabs*'(*x*) **fi**
  else ERROR(`*l'argument doit représenter un nombre*`)
  fi
end

Essayons de voir comment Maple calcule un sinus :

```
> eval(sin);
```

proc(*x*::*algebraic*) ... **end**

Zut...les procédures prédéfinies ne s'affichent pas. Mais on peut forcer leur affichage par une instruction ésotérique :

```
> interface(verboseproc=2);
```

Vous pouvez ensuite faire *eval*(*sin*); je ne le fais pas car l'affichage est trop long.

Noter que vous pourrez ainsi savoir comment Maple calcule *sin*(Pi) mais pas *sin*(1.5) car *sin*(1.5) est transmis à la procédure *evalf*. Regardons celle-ci :

```
> eval(evalf);
```

proc() **option** *builtin*, *remember*; 100 **end**

Là il n'y a aucun espoir : pour des raisons d'efficacité la procédure *evalf* fait partie du noyau de Maple, ce qui signifie qu'elle est écrite non pas en Maple mais en langage machine, et donc illisible.

Comme pour un tableau, une procédure est constituée de plusieurs composantes auxquelles on peut accéder avec *op*. Les opérandes 1,2,3,4,6 donnent respectivement les paramètres, variables locales, options, table des valeurs particulières, variables globales.

Exemple :

```
> op(4,eval(sin));
```

table([

$$\frac{3}{8}\pi = \frac{1}{2}\sqrt{2+\sqrt{2}}$$

$$\frac{5}{12}\pi = \frac{1}{4}\sqrt{6}\left(1+\frac{1}{3}\sqrt{3}\right)$$

$$\pi = 0$$

$$\frac{1}{12}\pi = \frac{1}{4}\sqrt{6}\left(1-\frac{1}{3}\sqrt{3}\right)$$

$$\frac{3}{10}\pi = \frac{1}{4}\sqrt{5}+\frac{1}{4}$$

$$\frac{2}{5}\pi = \frac{1}{4}\sqrt{2}\sqrt{5+\sqrt{5}}$$

$$0 = 0$$

$$\frac{1}{3}\pi = \frac{1}{2}\sqrt{3}$$

$$\frac{1}{8}\pi = \frac{1}{2}\sqrt{2-\sqrt{2}}$$

$$\frac{1}{5}\pi = \frac{1}{4}\sqrt{2}\sqrt{5-\sqrt{5}}$$

$$\frac{1}{4}\pi = \frac{1}{2}\sqrt{2}$$

$$I = I\sinh(1)$$

$$\frac{1}{2}\pi = 1$$

$$\frac{1}{10}\pi = \frac{1}{4}\sqrt{5}-\frac{1}{4}$$

$$\frac{1}{6}\pi = \frac{1}{2}$$

])