Structures de données avec Maple

Par Sadik BOUJAIDA

Introduction

Le typage des données sous Maple est trés varié. Il fournit tous les types utilisés dans les langages classiques et offre différentes méthodes de construction et de manipulation pour les types composites tels les les listes, tableaux et chaines de caractères. Maple en tant qu'interpreteur de commande n'offre pas de commandes de gestion de la mémoire (à la manière d'un langage compilé et même parfois interpreté), cette fonction est dévolue à son "ramasse–miettes" (gabrage collector).
Une conséquence de ce choix est que la déclaration des types de données, bien que possible avec Maple, n'est pas obligatoire. C'est au moment de l'initialisation d'une variable que son type est reconnu. En outre les régles de conversion de types sont trés permissives. Une variable qui, par exemple, à l'origine a reçu une donnée de type tableau, peut se voir changer de valeur en un nombre réel par une simple affectation. Bien que ceci puisse nuire à la robustesse et à la rigueur d'un programme écrit sous Maple, ce comportement est bien adapté à une utilisation intéractive.
Un autre aspect de la gestion des types de données sous Maple est la manière dont les variables sont évaluées à l'éxécution : certaines sont immédiatement remplacées par leurs contenus, alors que pour d'autres cette substitution ne se fait que jusqu'au dernier nom reçu par la variable.
Maple offre des procédures de contrôle du type de donnée d'une variable (type, whattype , ...) , ce qui est très utile pour effectuer des branchements par exemple. La procédure *convert* est specialisée dans la conversion explicite d'un type vers un autre, d'un tableau vers un ensemble, ou d'une écriture décimale vers une écriture en binaire par exemple. D'autres procédure permettent de sélectionner parmi plusieurs objets ceux répondant à certains critères, notament le type de donnée (*select*,*remove* , ...).

Types de données simples : les types booliens et les nombres.

Comme pour tout langage de programmation, le type boolien est bien présent, il prend les valeurs logiques *true* ou *false*. La maniplutaion des variables de type boolien sera traité en détail dans le paragraphe dédié aux instructions conditionnelles.

En plus des variables de type boolien, Maple gére nativement tous les types de nombres (entiers, rationnels, réels et complexes) sans autre limite pour leur taille que la quantité de mémoire disponible dans le système (pas de sous–typage short, long et autres ...). Chaque type de nombres peut être representé sous différents types de données (pour

répondre aux besoins en calcul formel). Une variable donnée, qui peut être l'un des argument d'une procédure peut être asservie à un type bien précis ou pour répondre à certaines conditions en utilisant la procédure *assume*.

Les nombres entiers

Les entiers (type *integer*), peuvent être de taille arbitrairemnt grande. il se subdivisent en plusieurs sous--type: *posint, negint*, respectivement pour entiers strictement positif ou negatif , et leurs négations *nonposint* et *nonnegint*.

```
> type(2,posint);type(-3,nonposint);type(0,negint);type(0,nonposint);
```

true
true
false
true

Les nombres rationnels

Les rationnels (type *rational*), sont manipulés avec leurs valeurs exactes sous forme de fractions, et non avec des valeurs approchées en virgules flottantes. Signalons qu'il existe un type *fraction* qui est en fait un rationnel écrit sous forme d'une fraction d'entiers qui n'est pas lui même un entier .

```
> 1/2+1/3;1/(5!);
```

$$\frac{5}{6}$$
$$\frac{1}{120}$$

```
> type(1/2,rational);type(0.5,rational);
```

true
false

```
> type(1/2,fraction);type(2,fraction);type(0.5,fraction);
```

true
false
false

Nombres réels

Pour les nombres réels, on dispose de plusieurs types qui se chevauchent dans leurs domaines de validité :

- ***realcons*** : pour toute constante réelle, même representée sous forme symbolique tels que pi.
- ***numeric*** : pour toute variable representée sous forme numérique.
- ***float*** : pour toute variable representant un nombre à virgule flottante.

```
> restart:
> type(Pi,realcons);type(Pi,numeric);type(Pi,float);
```

true
false
false

```
> type(.5,realcons);type(.5,numeric);type(.5,float);
```

true
true
true

```
> type(1/2,realcons);type(1/2,numeric);type(1/2,float);
```

true

true
false

la procédure evalf peut donner une valeurs approchée en virgules flotantes de n'importe quel nombre (réel ou complexe), avec une précision par défaut de 8 chiffres flottants. *evalf(a,n)* permet en outre d'évaluer le nombre a avec n chiffres flottants.

```
> evalf(Pi); evalf(exp(2*I*Pi/5));
```

$$3.141592654$$
$$0.3090169944 + 0.9510565163\,I$$

```
> evalf(Pi,75);
```

$$3.14159265358979323846264338327950288419716939937510582097494459230781640629$$

On peut fixer pour toute la session le nombre de chiffre flottant dans la représentation des floats grace à la variable d'environnement *Digits*.

```
> Digits:=25:
> evalf(Pi);
```

$$3.1415926535897932384626433$$

Nombres complexes

Les nombres complexes, comme pour les réels, peuvent être représentés sous différents types de données;

- ***complex*** : un complexe écrit obligatoirement sous forme algebrique $x+Iy$, où x et y sont des réels de type realcons, à signaler que si l'une des composantes x ou y est de type float, l'autre est automatiquement convertie en ce type.
- ***complexcons*** : constante complexe qui peut se presenter sous forme algebrique ou trigonometrique.

```
> type(exp(2*I*Pi/5),complex);type(exp(2*I*Pi/5),complexcons);
```

false
true

```
> exp(2*I*Pi/5);
```

$$e^{\frac{2}{5}\,I\pi}$$

evalc(z) permet d'ecrire z sous forme algebrique quelquesoit la représentation d'origine de z avec la particularité que tout nom intervenant dans la définition de z est considéré comme un réel. *Im(z)* et *Re(z)* remplissent le rôle que leurs noms indiquent, mais z doit être fourni sous forme algébrique.

```
> z:=exp(2*I*Pi/5)/(a+2*I);
```

$$z := \frac{e^{\frac{2}{5}\,I\pi}}{a + 2\,I}$$

```
> evalc(z);
```

$$\frac{\cos\left(\frac{2}{5}\,\pi\right) a}{a^2+4} + \frac{2\sin\left(\frac{2}{5}\,\pi\right)}{a^2+4} + I\left(\frac{\sin\left(\frac{2}{5}\,\pi\right) a}{a^2+4} - \frac{2\cos\left(\frac{2}{5}\,\pi\right)}{a^2+4}\right)$$

```
> Im(z);Im(evalc(z));
```

$$\Im\left(\frac{e^{\frac{2}{5}\,I\pi}}{a + 2\,I}\right)$$
$$\Im\left(\frac{\cos\left(\frac{2}{5}\,\pi\right) a}{a^2+4} + \frac{2\sin\left(\frac{2}{5}\,\pi\right)}{a^2+4} + I\left(\frac{\sin\left(\frac{2}{5}\,\pi\right) a}{a^2+4} - \frac{2\cos\left(\frac{2}{5}\,\pi\right)}{a^2+4}\right)\right)$$

```
> Z:=x+I*y;
```

$$Z := x + \mathrm{I}\, y$$

```
> Im(Z);
```

$$\Im(x + \mathrm{I}\, y)$$

```
> type(Z,complex);type(Z,complex(name));
```

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

```
> assume(x,real);assume(y,real);
> type(Z,complex);Im(Z);
```

$$\mathit{false}$$
$$y\sim$$

Séquences, Listes et ensembles:

Les séquences, listes et ensembles sont des objets composite, mais dont la gestion reste simple: ils obéissent aux même regles d'evaluation que les types numériques, substitution immédiate du nom d'une variable par son contenu.

Séquences

Une séquence est une liste (au sens littéral, non au sens Maple) d'objets de types quelconques, séparés par des virgules.

```
> S:=0,I,sin,matrix(2,2,1);
```

$$S := 0, \mathrm{I}, \sin, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

On peut accéder aux opérandes d'une séquence en utilisant l'opérateur d'indexation [ ]

```
> S[3];S[1..3];S[-1];S[-2..-1];
```

$$\sin$$
$$0, \mathrm{I}, \sin$$
$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$
$$\sin, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

la procédure *seq* sert à la construction de séquences structurées . Son utilisation peut être sous la forme

seq(f(k),k=n..m) ou **seq(f(k),k=obj)**

Dans le premier cas , la sequence *f(n),f(n+1),...,f(m)* sera construite en incrémentant la variable m, d'une unité jusqu'à atteindre m, m et n peuvent être des rationals.
Dans le second sera construite la séquence *f(op1),f(op2),...,f(opN)* où *op1,op2,...,opN* sont les opérandes de l'objet *obj*, *obj* étant une entité Maple quelconque.

```
> seq(k,k=1..10);seq(k,k=1/2..5);seq(k^2,k=a+b+c+d);seq(k,k="Hello World!");
```

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10$$
$$\frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \frac{7}{2}, \frac{9}{2}$$
$$a^2, b^2, c^2, d^2$$
$$\text{"H"}, \text{"e"}, \text{"l"}, \text{"l"}, \text{"o"}, \text{" "}, \text{"W"}, \text{"o"}, \text{"r"}, \text{"l"}, \text{"d"}, \text{"!"}$$

L'opérateur *$* permet aussi de construire des séquences simples, son utilisation est moins polyvalente que *seq*

```
> x$5;$1..10;
```

$$x, x, x, x, x$$

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

```
> x^k$k=1..10;
```

$$x, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9, x^{10}$$

```
> k$k=a+b+c+d;
```

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

Le type d'une séquence :

```
> whattype(S);
```

$$exprseq$$

La séquence vide est désigné par le mot clé *NULL,* utile pour initialiser une séquence qui va être remplie par une boucle ensuite.

On peut récupérer la séquence des opérandes de n'importe quels objet Maple avec la procédure *op* , ce qui expliquerait le comportement dans l'instruction *seq(f(k),k=obj)*

```
> x:=a+b+c+d;y:=a/b;z:=a^3;
```

$$x := a + b + c + d$$
$$y := \frac{a}{b}$$
$$z := a^3$$

```
> op(x);op(y);op(z);
```

$$a, b, c, d$$
$$a, \frac{1}{b}$$
$$a, 3$$

Listes

Une liste est une séquence Maple mise entre crochets. De ce fait on utilise souvent les constructeurs de séquences *seq* et *$* pour construire des listes.

```
> restart;
  S:=seq(x^k,k=1..10);L:=[seq(x^k,k=1..10)];
```

$$S := x, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9, x^{10}$$
$$L := [x, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9, x^{10}]$$

```
> L[5];L[-5..-1];
```

$$x^5$$
$$[x^6, x^7, x^8, x^9, x^{10}]$$

On peut accéder aux opérandes d'une liste en utilisant aussi la procédure *op*. Cette dernière renvoyerait une erreur si on l'utilise avec une séquence.

```
> op(5,L);
```

$$x^5$$

```
> op(4,S);
```

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

op(L) retourne la séquence des les opérandes de *L* :

```
> op(L);
```

$$x, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9, x^{10}$$

Le nombre d'opérandes d'une liste *L* est donné par *nops(L)*, une fois encore nops provoquera une erreur si elle est utilisée avec une séquence.

```
> nops(L);
```

$$10$$

```
> nops(S);
```

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

Petite astuce, si on tient à avoir le nombre d'opérandes d'une séquence.

```
> nops([S]);
```

$$10$$

N.B: *nops(obj)* retourne en fait le nombre d'opérandes de n'importe quel objet Maple (sauf une séquence bien sur).

Type d'une liste :

```
> whattype(L);
```

$$list$$

Ensembles

Un ensemble au sens Maple est une séquence mise entre accolades {}. La différence entre un ensemble et une liste tient esentiellement en deux points. Dans un ensemble il n'y a pas d'ordre préalable des opérandes (l'ordre d'entrée des opérandes, n'est pas forcément celui de la sortie), l'autre point est que les doublons sont automatiquement éléminé dans un ensemble.

```
> E:={1,(-1)^2,sin,matrix(2,2,1)};
```

$$E := \left\{1, \sin, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\right\}$$

```
> op(E);nops(E);
```

$$1, \sin, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$3$$

Type d'un ensemble :

```
> whattype(E);
```

$$set$$

Conversion entre types de données

La procédure *convert* permet de convertir un objet Maple quelconque en un autre avec un autre type mais construit à partir des mêmes opérandes.

```
> L;E;
```

$$[x, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9, x^{10}]$$

$$\left\{1, \sin, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\right\}$$

```
> convert(L,set);convert(E,list);
```

$$\{x^7, x^8, x^9, x^{10}, x^2, x^3, x^4, x^5, x^6, x\}$$

$$\left[1, \sin, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\right]$$

```
> convert(a+b+c+2*d,set);
```

$$\{c, b, a, 2\ d\}$$

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

$$a + b + c + d$$

Détail interessant , *convert* permet aussi de faire de la conversion entre systèmes de numération.

```
> convert(17092009,binary);convert(17092009,octal);convert(17092009,hex);
```

$$100000100110011010101001$$

$$101146651$$

Ou plus intéressant encore, puisque la sortie est presentée sous forme d'une liste, les coéfficients étant pris à partir des unités (de droite vers la gauche) :

```
> convert(17092009,base,2);convert(17092009,base,8);convert(17092009,base,16);
```

$$[1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1]$$
$$[1, 5, 6, 6, 4, 1, 1, 0, 1]$$
$$[9, 10, 13, 12, 4, 0, 1]$$

Tableaux et tables

Tableaux et tables sont des structures de données composites, capable de stocker des données discrètes, sur une ou deux dimensions pour les tableaux, ou avec un système clé/valeur, à la mainière d'un dictionnaire pour les tables.

Tableaux

Un tableau Maple peut être à une ou à deux dimensions. A la différence d'une liste où aucune indication sur la taille n'est fournie lors de la définition, il est nécéssaire de déclarer la taille d'un tableau. Son initialisation peut se faire lors de sa définition ou plus tard (grace à une boucle par exemple).

Pour la construction de tableaux, la syntaxe est assez permissive :

- **array(1..n, list_coeff):** tableau à une dimension avec n coéfficient, list_coeff, non obligatoire, devrait être une liste de taille inferieure ou égale à n formée d'objets Maple quelconques.
- **array(list_coeff) :** On omet de préciser la taille, mais elle est toujours obligatoire, elle est calculée à partir de la taille de list_coeff.
- **array(1..n,1..m,list_list_coeff) :** tableau à deux dimensions, n lignes et m colonnes. list_list_coeff, devrait être une liste de listes d'objets Maple, chacune représentant une ligne du tableau. La taille de chaque liste intérieure doit être inférieure ou égal à m, le nombre de ces listes doit être inférieur ou égal à n.
- **array(list_list_coeff) :** la taille est calculée à partir de list_list_coeff, la taille de la première liste donne le nombre de colonnes, le nombre de listes celui des lignes.

```
> T_one:=array(1..3,[1,Pi]);T_two:=array([[7,8],[1],[]]);
```

$$T\_one := \left[\begin{array}{ccc} 1 & \pi & T\_one_{3} \end{array}\right]$$

$$T\_two := \left[\begin{array}{cc} 7 & 8 \\ 1 & T\_two_{2,2} \\ T\_two_{3,1} & T\_two_{3,2} \end{array}\right]$$

On accéde aux coéfficients par leurs indices, y compris pour les initier par afféctation directe.

```
> T_one[1];T_one[3]:=0;
```

$$T\_one_{3} := 0$$

```
> print(T_one);
```

$$\left[\begin{array}{ccc} 1 & \pi & 0 \end{array}\right]$$

```
> T_two[2,2]:=0:T_two[3,1]:=0:T_two[3,2]:=0:
> print(T_two);
```

$$\begin{bmatrix} 7 & 8 \\ 1 & 0 \\ 0 & 0 \end{bmatrix}$$

Une bizarrerie liée au mode d'évaluation des tableaux, qu'on abordera plus en avant.

```
> whattype(T_one);op(T_one);
```

$$symbol$$

$$\begin{bmatrix} 1 & \pi & 0 \end{bmatrix}$$

```
> whattype(eval(T_one));op(eval(T_one));
```

$$array$$

$$1..3, [1 = 1, 2 = \pi, 3 = 0]$$

Tables

Une table Maple, est ce qu'on appelle parfois dictionnaire sous d'autres langages de programmation. C'est une structure composite capable de stocker des données en utilisant un systéme d'indéxation clé/valeur. A la différence d'un tableau, il n'y a pas d'obligation de préciser la taille d'une table. Une table est de fait de taille indéterminée.

Pour créer une table :

```
> Tab_1:=table([un=sup,deux=spe]);
```

$$Tab\_1 := table([deux = spe, un = sup])$$

```
> Tab_2:=table([sup,spe]);
```

$$Tab\_2 := table([1 = sup, 2 = spe])$$

```
> Tab_3:=table([(1,1)=1,(1,2)=0,(2,1)=0,(2,2)=1]);
```

$$Tab\_3 := table([(1,1) = 1, (1,2) = 0, (2,1) = 0, (2,2) = 1])$$

Pour créer la table vide :

```
> table();
```

$$table([])$$

Le fait d'utiliser un nom T suivi de l'opérateur d'indexation $[$ (quelque chose comme $T[a]$) dans une affectation, crée automatiquement une table de nom T , et initie la clé a avec la valeur qu'on a affecté à $T[a]$. Souvent on utilise l'indexation pour définir des termes d'une suites, on crée en fait une table.

```
> T[1]:=0;
```

$$T_1 := 0$$

```
> print(T);
```

$$table([1 = 0])$$

Une autre utilisation des tables dans Maple, consiste en la possibilité d'associer à une procédure une table, dite *table remember*, pour stocker les valeurs calculées de cette procédure. Fonctionnalité surtout utilisée avec des procédures récursives.

Evaluation dans Maple

Evaluation des variables

Les variables simples, les séquences et donc les listes et les ensembles sont immédiatement remplacés par leurs valeurs quand on fait intervenir leurs noms dans une expression. Les tableaux, les tables (et les procédures) ne sont évalués que jusqu'à leurs derniers noms.

Par exemple si on définit les deux variables:

```
> a:=Pi;A:=array(1..2,[1,1]);
```

$$a := \pi$$
$$A := \begin{bmatrix} 1 & 1 \end{bmatrix}$$

Un appel de ces variables par leurs noms donne :

```
> a;A;
```

$$\pi$$
$$A$$

a est completement évaluée, alors que la valeur retournée pour le tableau est son derniers nom A.

```
> B:=A;C:=B;
```

$$B := A$$
$$C := A$$

```
> C;
```

$$A$$

Pour faire référence au contenu de A et non simplement à son nom, il faut utiliser la procédure d'evalution *eval*

```
> eval(A);
```

$$\begin{bmatrix} 1 & 1 \end{bmatrix}$$

Ce comportement s'explique par le fait, que lors de l'évaluation d'une expression qui fait intervenir plusieurs variables, Maple effectue d'abords des simplifications automatiques selon les régles applicables au domaine des variables en jeu (commutativité, associativité, calcul algebrique ...). Ces regles seront donc appliquées uniquement sur les noms des variables composites complexes, exigentes en mémoire, au lieu de le faire sur leurs contenus.

Un autre aspect de l'évaluation des variables complexes sous Maple, est que le nom d'un tableau (ou d'une table) A est un pointeur qui pointe vers son contenu. lorsque on donne un autre nom B à A, les noms A et B continuent de pointer vers le même objet en mémoire. Si on change B, A va aussi changer. Pour une variable simple, lors d'une deuxième affectation, une copie du contenu est crée pour le nouveau nom, de sorte que les deux variables deviennent complétement indépendantes.

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

$$b := \pi$$

```
> b;a;
```

$$\pi$$
$$\pi$$

```
> b:=0;a;
```

$$b := 0$$
$$\pi$$

```
> C;eval(A);
```

$$A$$
$$\begin{bmatrix} 1 & 1 \end{bmatrix}$$

```
> C[1]:=0;eval(A);
```

$$C_1 := 0$$
$$\begin{bmatrix} 0 & 1 \end{bmatrix}$$

Si on tient à créer une copie autonome d'un objet complexe, il faut le demander explicitement en utilisant *copy* .

```
> eval(A);E:=copy(A);E[1]:=1;eval(E);eval(A);
```

$$\begin{bmatrix} 0 & 1 \end{bmatrix}$$
$$E := \begin{bmatrix} 0 & 1 \end{bmatrix}$$
$$E_1 := 1$$
$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

Retarder l'évaluation d'une variable

On peut pour une variable donnée (même de type simple), retarder son évaluation en mettant son nom entre quotes ' '.

```
> a;'a';
```

$$\pi$$
$$a$$

Ceci permet par exemple de libérer le nom d'une variable qui a déja reçu une valeur par affectation, en lui affectant à nouveau son nom.

```
> X:=1;
```

$$X := 1$$

```
> X:='X';X;
```

$$X := X$$
$$X$$

Avec plusieurs niveaux de quotations chaque évaluation diminue la protection d'un niveau.

```
> e:=''a'';f:=e;
```

$$e := 'a'$$
$$f := a$$

Structure interne d'une expression Maple :

Une expression Maple est entièrement déterminée par son type et par la séquences de ses opérandes, qui elles même sont des expressions. Une expression est donc construite de façon récursive sous forme d'un arbre, avec des sommets qui sont les types des variables intermédiaires et les terminaisons des branches sont les opérandes élémentaires.

```
> expr:=x+y*z;
```

$$expr := x + y\,z$$

en respectant les regles de précédences des opérateurs, l'expression *expr* est representée par l'arbre :

On peut demander le type d'une expression expr par l'instruction *whattype(expr)* ou *op(0, expr)*. La séquence complète de ses opérandes est données par *op(expr)*. la k eme opérande est désignée par *op(k,expr)*. la h eme opérande de la k eme est désignée par op([k,h],expr), ainsi de suite.

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

$$`+`$$
$$`+`$$

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

$$x, y\,z$$

$$x$$
$$x, y\,z$$

```
> op([2,1],expr);
```

$$y$$

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

$$`*`$$

On voit ainsi que les séquences représentent la structure de donnée la plus élémentaire dans Maple.

Pour un tableau (une table ou une procédure aussi)

```
> Tab:=array(1..2,[0,1]);
```

$$Tab := \left[\begin{array}{cc} 0 & 1 \end{array}\right]$$

N'oublions pas, un tableau est évalué jusqu'à son dernier nom.

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

$$symbol$$

Il faut plutôt faire :

```
> op(0,eval(Tab));
```

$$array$$

```
> op(eval(Tab));
```

$$1..2, [1 = 0, 2 = 1]$$

Les procédures *map*, *select* et *remove*

select permet de séléctionner, parmi les opérandes d'une expression, celles qui répondent à un certain critère .

select(fct_test, expr), fct_test est une procédure qui doit retourner forcément une valeurs logique (true ou false). Chaque opérande de l'expression est testé par *fct_test*, ne sont retenues que celles auquels la réponse est *true,* Maple construit alors un nouvel objet de même type que *expr* avec les opérandes retenues.

Liste des nombres premiers inférieur à 100 :

```
> select(isprime,[2,seq(2*k+1,k=1..49)]);
```

$$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]$$

Pour éléminer d'une liste toutes les opérandes qui sont nulles :

```
> select(x->if x<>0 then true else false fi,[1,0,2,0,9,0,0,0,8,7,6,5,0,0,1]);
```

$$[1, 2, 9, 8, 7, 6, 5, 1]$$

remove agit de la même façon mais ne retient que les opérandes qui ne répondent pas au test.

map, bien utile, permet d'appliquer une procédure à toutes les opérandes d'une expression.

map(fct,expr) : *fct* étant une procédure quelconque qui pend un seul argument, un objet de même type que *expr* est retourné avec les opérandes de *expr* auquelles est appliquée la procédure *fct*.

map(fct,expr,arg2,arg3,...,argN) : applique la procédure *fct* qui prend cette fois *N* argument, aux opérandes de *expr* en tant que premier argument, les autres arguments *arg2,..,argN* devant être précisé explicitement .

```
> map(x->x^2,a+b+c+d);
```

$$a^2 + b^2 + c^2 + d^2$$

```
> map((x,y)->(x+y)^2,a+b+c+d,1);
```

$$(a+1)^2 + (b+1)^2 + (c+1)^2 + (d+1)^2$$

```
> map(convert,[629,42,6598,27,9,47,820],binary);
```

[1001110101, 101010, 1100111000110, 11011, 1001, 101111, 1100110100]

Les chaines de charactères

Une chaine de charactère (type *string)* est une suite de charactères qui n'a d'autre signification qu'elle même, elle ne peut faire l'objet d'une affectation et sera toujours évaluée en elle même (d'après l'aide Maple).

les chaines de charactères seront définies comme des suites de charactères quelconques mises entre double-quotes. Elles serviront en général sous Maple à afficher des messages. Elles deviendront en outre indispensable dés qu'il s'agit d'écrire des programmes pour la manipulation de texte, ce qui n'est pas forcément le terrain de prédilection de Maple.

Définition d'une chaine de charactère

```
> Strg:="Une chaine qui se termine avec des charactères spéciaux /, $ et &";
```

Strg := "Une chaine qui se termine avec des charact□res sp□ciaux /, $ et &"

```
> whattype(Strg);
```

string

Une chaine de charactère est un type indexé (en fait un tableau dans certains langage), on peut accéder aux charactères d'une chaine par leurs indices

```
> Strg[1];Strg[-1];Strg[1..10];Strg[-9..-1];
```

"U"
"&"
"Une chaine"
"/, $ et &"

On peut chercher un motif dans une chaine de charactères en utilisant *searchtext* (indifférence à la casse) ou *SearchText* (respect de la casse). La valeur retournée est l'indice du premier charactère du motif dans la chaine si le motif est retrouvé dans la chaine, 0 sinon.

```
> searchtext(cde,"abcdefgh");SearchText(abc,"Abcdefgh");
```

3
0

On peut concaténer plusieurs chaine de charactères avec cat

```
> Strg_1:="Une première chaine"; Strg_2:=" et une deuxième";
```

Strg_1 := "Une premi□re chaine"
Strg_2 := " et une deuxi□me"

```
> cat(Strg_1,Strg_2);
```

"Une premi□re chaine et une deuxi□me"

cat peut servir pour formater une sortie texte basique (voir les fonctions *printf*, *fprintf* et *sprintf* pour un formatage plus élaboré, à la manière du langage C)

```
> for i to 5 do cat("La valeur de i est ",i) od;
```

"La valeur de i est 1"
"La valeur de i est 2"
"La valeur de i est 3"
"La valeur de i est 4"
"La valeur de i est 5"

Fonction intéressante, on peut convertir une chaine de charactère en une liste, la liste des codes héxadécimaux (exprimé en décimal !) des charactères de la chaine

```
> L:=convert(Strg,bytes);
```

L := [85, 110, 101, 32, 99, 104, 97, 105, 110, 101, 32, 113, 117, 105, 32, 115, 101, 32, 116, 101, 114, 109, 105, 110, 101, 32, 97, 118, 101, 99, 32, 100, 101, 115, 32, 99, 104, 97, 114, 97, 99, 116, 232, 114, 101, 115, 32, 115, 112, 233, 99, 105, 97, 117, 120, 32, 47, 44, 32, 36, 32, 101, 116, 32, 38]

La réciproque, la liste convertie en une chaine, avec la même syntaxe.

```
> convert(L,bytes);
```

"Une chaine qui se termine avec des charact□res sp□ciaux /, $ et &"

La liste des charactères de la table de code utilisée par le système

```
> L_char:=convert([$1..255],bytes);
```

L_char := "□□□□□□□□
□□
□□□□□□□□□□□□□□□□□□□□ !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]
^_abcdefghijklmnopqrstuvwxyz{|}
~□□
□□□"

Stocker tout dans une table indéxée par les codes des charactères

```
> Tab_char:=table(convert(L_char,list));
```

Tab_char := *table*([120 = "x", 241 = "□", 121 = "y", 1 = "□", 242 = "□", 122 = "z", 2 = "□", 243 = "□", 123 = "{", 3 = "□", 244 = "□", 124 = "|", 4 = "□", 245 = "□", 125 = "}", 5 = "□", 246 = "□", 126 = "~", 6 = "□", 247 = "□", 127 = "□", 7 = "□", 248 = "□", 128 = "□", 8 = "□", 249 = "□", 129 = "□", 9 = "   ", 250 = "□", 130 = "□", 10 = "
", 251 = "□", 131 = "□", 11 = "□", 252 = "□", 132 = "□", 12 = "□", 253 = "□", 133 = "□", 13 = "
", 254 = "□", 134 = "□", 14 = "□", 255 = "□", 135 = "□", 15 = "□", 136 = "□", 16 = "□", 137 = "□", 17 = "□", 138 = "□", 18 = "□", 139 = "□", 19 = "□", 140 = "□", 20 = "□", 141 = "□", 21 = "□", 142 = "□", 22 = "□", 143 = "□", 23 = "□", 144 = "□", 24 = "□", 145 = "□", 25 = "□", 146 = "□", 26 = "□", 147 = "□", 27 = "□", 148 = "□", 28 = "□", 149 = "□", 29 = "□", 150 = "□", 30 = "□", 151 = "□", 31 = "□", 152 = "□", 32 = " ", 153 = "□", 33 = "!", 154 = "□", 34 = """, 155 = "□", 35 = "#", 156 = "□", 36 = "$", 157 = "□", 37 = "%", 158 = "□", 38 = "&", 159 = "□", 39 = "'", 160 = "□", 40 = "(", 161 = "□", 41 = ")", 162 = "□", 42 = "*", 163 = "□", 43 = "+", 164 = "□", 44 = ",", 165 = "□", 45 = "-", 166 = "□", 46 = ".", 167 = "□", 47 = "/", 168 = "□", 48 = "0", 169 = "□", 49 = "1", 170 = "□", 50 = "2", 171 = "□", 51 = "3", 172 = "□", 52 = "4", 173 = "□", 53 = "5", 174 = "□", 54 = "6", 175 = "□", 55 = "7", 176 = "□", 56 = "8", 177 = "□", 57 = "9", 178 = "□", 58 = ":", 179 = "□", 59 = ";", 180 = "□", 60 = "<", 181 = "□", 61 = "=", 182 = "□", 62 = ">", 183 = "□", 63 = "?", 184 = "□", 64 = "@", 185 = "□", 65 = "A", 186 = "□", 66 = "B", 187 = "□", 67 = "C", 188 = "□", 68 = "D", 189 = "□", 69 = "E", 190 = "□", 70 = "F", 191 = "□", 71 = "G", 192 = "□", 72 = "H", 193 = "□", 73 = "I", 194 = "□", 74 = "J", 195 = "□", 75 = "K", 196 = "□", 76 = "L", 197 = "□", 77 = "M", 198 = "□", 78 = "N", 199 = "□", 79 = "O", 200 = "□", 80 = "P", 201 = "□", 81 = "Q", 202 = "□", 82 = "R", 203 = "□", 83 = "S", 204 = "□", 84 = "T", 205 = "□", 85 = "U", 206 = "□", 86 = "V", 207 = "□", 87 = "W", 208 = "□", 88 = "X", 209 = "□", 89 = "Y", 210 = "□", 90 = "Z", 211 = "□", 91 = "[", 212 = "□", 92 = "\", 213 = "□", 93 = "]", 214 = "□", 94 = "^", 215 = "□", 95 = "_", 216 = "□", 96 = "`", 217 = "□", 97 = "a", 218 = "□", 98 = "b", 219 = "□", 99 = "c", 220 = "□", 100 = "d", 221 = "□", 101 = "e", 222 = "□", 102 = "f", 223 = "□", 103 = "g", 224 = "□", 104 = "h", 225 = "□", 105 = "i", 226 = "□", 106 = "j", 227 = "□", 107 = "k", 228 = "□", 108 = "l", 229 = "□", 109 = "m", 230 = "□", 110 = "n", 231 = "□", 111 = "o", 232 = "□", 112 = "p", 233 = "□", 113 = "q", 234 = "□", 114 = "r", 235 = "□", 115 = "s", 236 = "□", 116 = "t", 237 = "□", 117 = "u", 238 = "□", 118 = "v", 239 = "□", 119 = "w", 240 = "□"])

(Sous Microsft Windows, faites un **alt+code du charactère** dans Notepad pour vérifier)

```
> Tab_char[k]$k=65..90;
```

"A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"

Signification des quotes

Il y a trois types de quotations, chacune à une signification spéciale: la quote ', la backquote ` et la double quote ".

La quote sert à retarder l'évaluation d'une variable, comme on l'a vu.

La double quote sert à construire des chaines de charactères.

La backquote sert à construire des noms de variables. Elle permet d'utiliser des charactères illégitime dans les noms, tel l'espace

```
> Un nom:=NULL;
```

Error, missing operator or `;`

```
> `Un nom`:=NULL;
```

$$Un\ nom := $$

Il faut faire la différence entre un nom de variable et une chaine de charactère. Un nom pointe vers une valeur qui n'est pas le nom lui même, une chaine de charactère n'a d'autre valeur qu'elle même.

Enfin signalons que l'on peut provoquer l'affichage d'un message texte en utilisant *print* , et que *ERROR* est une fonction spécialisée dans la gestion des erreurs, et qui entre autre provoque l'affichage d'un message d'erreur et l'arrêt immediat de l'exécution d'un programme.

```
> print("Hello World!");
```

"Hello World!"

```
> ERROR("Ceci est un message d'erreur");
```

Error, Ceci est un message d'erreur

Piles et files :

Les piles et les files sont des structures de données linéaires pouvant stocker des données de façon temporaire et les restituer dans un certain ordre. A l'inverse d'un tableau (ou d'une table) les données ne sont pas indexées, ce qui en fait des structures simple à gérer et plutôt bien adaptées à certain problèmes.

Dans une file c'est une logique de premier arrivé -- premier servi qui prédomine (first in -- firt out, ou FIFO), dans une file c'est plutôt dernier arrivé -- premier servi (last in -- first out, ou LIFO).

Maple gére les piles et les files au travers des packages respectifs **stack** et ***queue*** .

Cependant le chargement simultané de ces deux packages pose un problème.

```
> with(stack);
```

[*depth, empty, new, pop, push, top*]

```
> with(queue);
```

Warning, the names empty and new have been redefined

Warning, the protected name length has been redefined and unprotected

[*clear, dequeue, empty, enqueue, front, length, new, reverse*]

Comme on le constate lors du chargement du package queue en second lieu, Maple averti de la redéfinition des procédures empty et new, qui font partie aussi du package stack. Il est alors préferable en cas de besoin d'utilisation simultanées des deux structures de non pas charger les deux packages, mais d'appeler chaque fonction par son raccourci (sans avoir besoin de charger les packages). Par exemple pour appeler la fonction new qui sert à construire une nouvelle pile qu'on va appeler P, il suffit d'utiliser la commande stack[new](P).

Les piles

Syntaxes et explications

- stack[new]() ou stack[new](elt1,elt2,...,eltN): définit une nouvelle pile (vide ou contenant les éléments elt1,el2,...,eltN, eltN, avec elt1 au sommet de la pile). On peut donner un nom à la nouvelle pile par simple affectation.
- stack[push](elt,P): ajouter l'élément elt au sommet de la pile P.

- stack[pop](P), stack[top](P) : creuser un élément dans la pile P, en l'enlevant de la pile dans le premier cas, en le conservant dans le deuxième.
- stack[empty](P) : teste si la pile est vide, retourne une valeur logique true ou false.
- stack[depht](P) : retourne la profondeur 'le nombre d'éléments) de la pile.

Noter que pour vider une pile, il suffit de la reinitialiser avec une pile vide.

Exemple :

On crée une pile vide de nome Eleves:

```
> Eleves:=stack[new]();
```

$$Eleves := table([0 = 0])$$

```
> stack[empty](Eleves);
```

$$true$$

On empile des éléments dans la pile en utilisant une boucle :

```
> for k in [amine,salma,safsaf] do stack[push](k,Eleves) od;
```

$$amine$$
$$salma$$
$$safsaf$$

```
> stack[empty](Eleves);
```

$$false$$

```
> eval(Eleves);
```

$$table([0 = 3, 1 = amine, 2 = salma, 3 = safsaf])$$

```
> stack[pop](Eleves);
```

$$safsaf$$

```
> eval(Eleves);
```

$$table([0 = 2, 1 = amine, 2 = salma])$$

Notez bien que, **à cause** de l'implémentation des piles comme des tables , on peut accéder aux éléments par leurs indices (en pis des traditions dans l'univers de la programmation), l'élément de la table indexé par 0 étant la profondeur de la pile.

```
> Eleves[0];Eleves[1];
```

$$2$$
$$amine$$

Type d'une pile :

```
> whattype(eval(Eleves));
```

$$table$$

```
> type(Eleves,stack);
```

$$true$$

Quand même ...

Les files:

Syntaxes des differentes commandes du package :

- queue[new]() ou queu[new](elt1,elt2,...,eltN) : comme pour les piles. elt1 sera stocké en tête de la queue
- queue[enqueue](Q,elt) : placer l'élément elt dans la file Q, noter que l'ordre des arguments est inversé par rapport à une pile.
- queue[dequeue](Q) : enlever l'élément en tête de la file Q, la commande retoutne le nome de l'élément enlevé.
- queue[front](Q) : retourne le nom de l'élément en tête de la file Q sans l'enlever.
- queue[length](Q) : retourne le nombre d'éléments dans la file Q.

- queue[empty](Q) : teste si la file est vide.
- queue[clear](Q) : vide la file Q.
- queue[reverse](Q) : inverse l'ordre des éléments dans la file Q.

Les opérations dequeue, front et reverse si elle sont effectuées sur pile vide provoquent une erreur, avant de les utiliser dans un programme il faut toujours tester si la file n'est pas vide avec empty.

Exemple

```
> Alphabet:=queue[new]();
```

$$Alphabet := table([0 = 0])$$

```
> for k in "ABCDE" do queue[enqueue](Alphabet,k) od;
```

$$k := \text{"A"}$$
$$\text{"A"}$$
$$k := \text{"B"}$$
$$\text{"B"}$$
$$k := \text{"C"}$$
$$\text{"C"}$$
$$k := \text{"D"}$$
$$\text{"D"}$$
$$k := \text{"E"}$$
$$\text{"E"}$$

```
> queue[front](Alphabet);
```

$$\text{"A"}$$

```
> eval(Alphabet);
```

$$table([0 = 5, 1 = \text{"A"}, 2 = \text{"B"}, 3 = \text{"C"}, 4 = \text{"D"}, 5 = \text{"E"}])$$

```
> queue[reverse](Alphabet);
> eval(Alphabet);
```

$$table([0 = 5, 1 = \text{"E"}, 2 = \text{"D"}, 3 = \text{"C"}, 4 = \text{"B"}, 5 = \text{"A"}])$$

```
> queue[clear](Alphabet);
```

$$5$$

```
> eval(Alphabet);
```

$$table([0 = 0])$$

Procédures de contrôle de typage :

Le service offert par les procédure *type* et *whattype* est très précieux. Ils permettent de controler le type de donnée d'une variable. Ainsi elles peuvent intervenir des branchements conditionnels (instruction if ou while).

- **type(obj,struct)** : retourne *true* si **obj** est de type **struct**, *false* sinon.
- **whattype(obj)** : retourne le type de donnée de l'objet **obj.**

Signalons que type permet de spécialiser la requete dans le cas où une variable fait partie de plusieurs domaine de typage. whattype elle se contente de retourner le type le plus adapté d'une variable donnée.

```
> type(-2,integer);type(-2,negint);type(-2,numeric);type(-2,realcons);
```

$$true$$
$$true$$
$$true$$
$$true$$

```
> whattype(-2);
```

$$integer$$

```
> A:=table([]);
```

$$A := table([\ ])$$

```
> whattype(A);
```

symbol

```
> whattype(eval(A));
```

table

```
> P:=stack[new]();
```

$$P := table([0 = 0])$$

```
> type(eval(P),table);
```

true

```
> type(eval(P),stack);
```

true

```
> whattype(eval(P));
```

table

```
> f:=x->x^2;
```

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

```
> whattype(f);
```

symbol

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

procedure

```
> whattype('f(x)');
```

function

```
> whattype(g(x)); #aucune procédure de nom g n'a été définie
```

function