STRUCTURATION ET TYPOLOGIE DES EXPRESSIONS

Bernard Dupont

Bernard.Dupont@univ-lille1.fr

Tout objet syntaxiquement valide est par définition une expression qui a une nature informatique interne précise pour des raisons de stockage en mémoire d'une part et de gestion des propriétés à fins de manipulations ultérieures d'autre part.
Le stockage d'une expression se fait par déstructuration raisonnée de l'objet en éléments élémentaires (section 1).
Sa gestion est commandée par le type qui lui est attribué. Une expression est en effet classée dans un groupe principal - voire dans un groupe principal et des groupes secondaires - qui ont des propriétés spécifiques (section 2). Grâce à ce classement, l'entier naturel 7 et le message "Erreur de saisie" n'ont pas le même type informatique et les règles de manipulation de ces deux expressions ne peuvent pas être les mêmes. Il faut évidemment apprendre à connaître le type d'une expression pour inférer la manière dont elle est gérée. C'est particulièrement vrai en programmation où il est souvent crucial de déclarer le type des variables entrant dans une procédure afin d'éviter des déboires (pourquoi s'obstiner à calculer la dérivée seconde d'un message d'erreur?).

Structuration d'une expression

Toute expression valide, qu'elle soit mathématique, littérale ou autre (graphique, photographie, animation, etc.), est immédiatement structurée suivant une logique informatique précise. La commande **dismantle** permet d'accéder à la structure Maple d'une expression. Le nombre 12.327 est traduit comme suit :

```
> restart;
  dismantle(12.327);

FLOAT(3): 12.327
   INTPOS(2): 12327
   INTNEG(2): -3
```

et le polynôme $a\,x^2 + b\,x + c$ par :

```
> dismantle(a*x^2+b*x+c);

SUM(7)
   PROD(5)
      NAME(4): a
      INTPOS(2): 1
      NAME(4): x
      INTPOS(2): 2
   INTPOS(2): 1
```

```
   PROD(5)
      NAME(4): b
      INTPOS(2): 1
      NAME(4): x
      INTPOS(2): 1
   INTPOS(2): 1
   NAME(4): c
   INTPOS(2): 1
```

En fait, Maple décompose par étapes successives une expression en éléments appelés **opérandes** reliés par des **opérateurs** jusqu'à obtention d'objets élémentaires. Reprenons le cas de l'expression polynômiale $a\,x^2+b\,x+c$. A un premier niveau, elle est la somme de 3 termes comme on le constate par l'instruction **op(a*x^2+b*x+c)**. En règle générale, si **xp** est une expression, l'instruction **op(xp)** renvoie la liste de tous les opérandes de niveau 1 et l'instruction **nops(xp)** le nombre des opérandes de niveau 1. Ici, on a :

```
> op(a*x^2+b*x+c);#décomposition en opérandes de niveau 1
  nops(a*x^2+b*x+c);#nombre d'opérandes de niveau 1
```

$$a\,x^2,\ b\,x,\ c$$

$$3 \qquad \textbf{(1.1)}$$

On peut isoler le *n*-ième opérande de niveau 1 d'une expression **xp** avec **op(n,xp)** avec **n** compris entre 1 et **nops(xp)** :

```
> op(1,a*x^2+b*x+c);#extraction du premier opérande de niveau 1
  op(2,a*x^2+b*x+c);#extraction du second opérande de niveau 1
  op(3,a*x^2+b*x+c);#extraction du troisième opérande de niveau 1
  op(4,a*x^2+b*x+c);#le message d'erreur signale que l'expression
  n'a pas de 4° opérande de niveau 1.
```

$$a\,x^2$$

$$b\,x$$

$$c$$

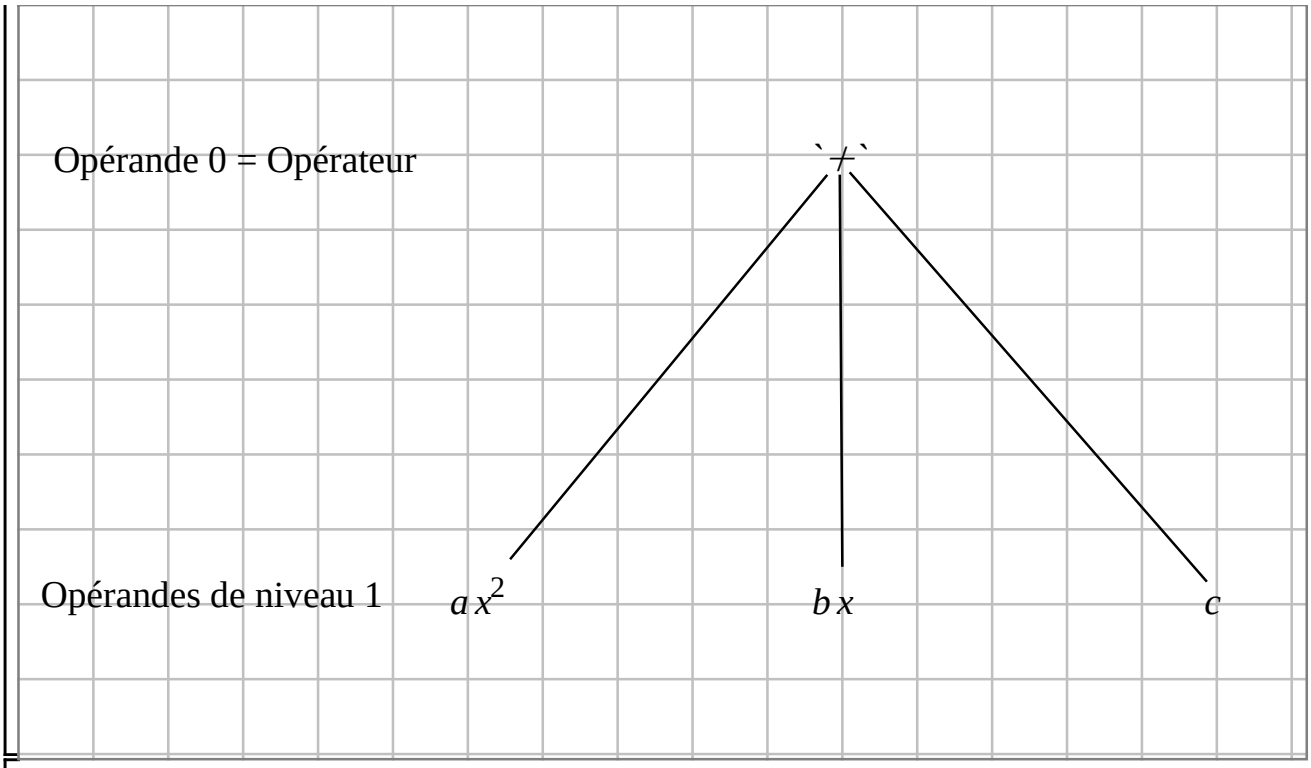
```
Error, improper op or subscript selector
```

L'opérateur qui structure les opérandes de niveau 1 correspond à un opérande d'indice 0. L'instruction **op(0,xp)** renvoie l'opérateur principal qui est généralement le type informatique de l'expression. Dans le cas du polynôme, on constate sans surprise que les trois opérandes de niveau 1 sont liés par l'opérateur d'addition :

```
> op(0,a*x^2+b*x+c);
```

$$`+` \qquad \textbf{(1.2)}$$

Le graphique suivant illustre ce principe hiérarchisant l'opérateur et les opérandes de niveau1.



Bien entendu, il est possible qu'un opérande de niveau 1 soit lui-même une expression dont on peut connaître le type et les opérandes. Les opérandes d'un opérande de niveau 1 sont logiquement appelés des opérandes de niveau 2. Dans l'exemple, le monôme $a\,x^2$ est une expression opérande de niveau 1 qui a deux opérandes de niveau 2 liés par l'opérateur de multiplication :

```
> op(op(1,a*x^2+b*x+c));#décomposition en opérandes du premier
  opérande de niveau 1
  nops(op(1,a*x^2+b*x+c));#nombre d'opérandes du premier opérande
  de niveau 1
  op(1,op(1,a*x^2+b*x+c));#premier opérande de niveau 2 du
  premier opérande de niveau 1
  op(2,op(1,a*x^2+b*x+c));#second opérande de niveau 2 du premier
  opérande de niveau 1
  op(0,op(1,a*x^2+b*x+c));#opérateur du premier opérande de
  niveau 1
```

$$a, x^2$$
$$2$$
$$a$$
$$x^2$$
$$\text{`*`}$$

(1.3)

Tant qu'un sous-opérande est une expression non élémentaire, on peut continuer à le décomposer. Ici, on décompose x^2 :

```
> op(op(2,op(1,a*x^2+b*x+c)));#opérandes de niveau 3 du second
  opérande de niveau 2 du premier opérande de niveau 1
  op(1,op(2,op(1,a*x^2+b*x+c)));#premier opérande de niveau 3 du
```

```
second opérande de niveau 2 du premier opérande de niveau 1
op(2,op(2,op(1,a*x^2+b*x+c)));#second opérande de niveau 3 du
second opérande de niveau 2 du premier opérande de niveau 1
op(0,op(2,op(1,a*x^2+b*x+c)));#opérateur du second opérande de
niveau 2 du premier opérande de niveau 1
```

$x, 2$

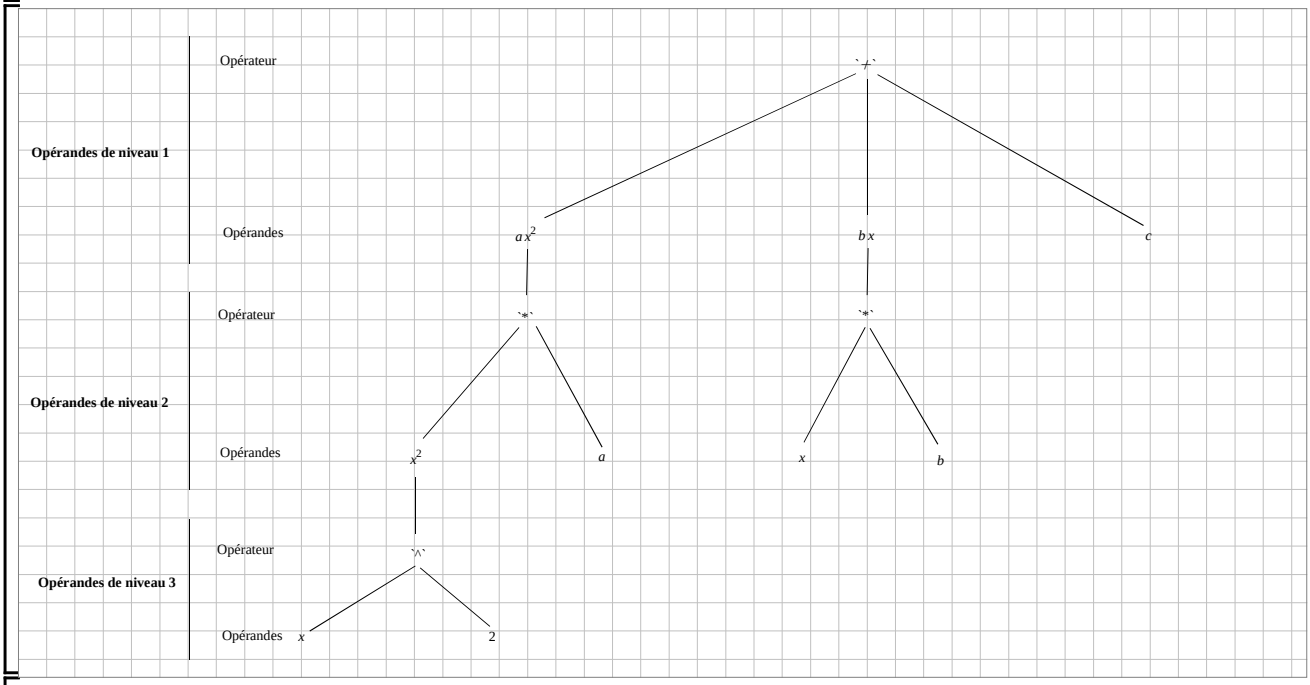
x

2

`^`

(1.4)

Au final, toute expression bien formée est décomposée en couples (opérateur, opérandes) jusqu'à ce que le dernier opérande à traiter soit un élément simple. Le polynôme de l'exemple est analysée de la manière suivante :



Signalons qu'il existe une syntaxe allégée de l'instruction **op** pour accéder aux sous-opérandes : ainsi, **op([m,n,p],xp)** extrait le p-ième opérande de niveau 3 du n-ième opérande de niveau 2 du m-ième opérande de niveau 1.

```
> op([1,2,1],a*x^2+b*x+c);#premier opérande de niveau 3 du second
  opérande de niveau 2 du premier opérande de niveau 1
```

x

(1.5)

A partir du moment où on peut accéder à un opérande particulier d'une expression, il devient possible de le modifier avec l'instruction **subs**. Cette technique s'utilise pour simplifier un résultat. Supposons par exemple qu'on désire transformer l'égalité $Y = \frac{A}{1-c}$ en $Y = \frac{A}{s}$ (Y est le revenu d'équilibre, A la dépense autonome et $s = 1 - c$ la propension à épargner des ménages).

```
> Eqc:=Y=A/(1-c);#Eqc est l'équation du revenu d'équilibre
  keynésien en fonction de la propension à consommer
  Eqs:=subs(op([2,2,1],Eqc)=s,Eqc);#Eqc est l'équation du revenu
```

d'équilibre keynésien en fonction de la propension à épargner

$$Eqc := Y = \frac{A}{1 - c}$$

$$Eqs := Y = \frac{A}{s} \quad \textbf{(1.6)}$$

Typologie des expressions

On vient de voir que la "tête pensante" d'une expression est un opérateur qui coordonne des opérandes. Dans la plupart des cas, l'opérateur agissant sur les opérandes de niveau 1 définit le type informatique de l'expression. Ainsi, l'instruction **op(0,xp)** donne généralement le type de l'expression **xp**. Il existe cependant des exceptions de taille : les fonctions, les séries et les noms indicés. Comme l'appartenance à un type donne l'étendue des manipulations possibles (et, par différence, l'étendue des manipulations impossibles), il faut, surtout en programmation, connaître le type exact de toutes les expressions en jeu. Les instructions **whattype** et **type** répondent à ce besoin. Elles sont présentées dans la sous-section 1.

La liste des types gérés par Maple est donnée dans la fiche dédiée du système d'aide :

```
> ?type
```

On recense plus de 200 types différents! La sous-section 2 est consacrée aux types les plus courants rencontrés en économie en mettant l'accent sur leurs propriétés majeures.

Reconnaissance des types et sous-types : whattype et type

La commande **whattype(xp)** permet de connaître le type principal de l'expression valide **xp**. Par exemple, si on écrit le polynôme de degré 3 : $ax^3 + bx^2 + cx + d$, on se rend compte que Maple le considère avant tout comme une expression du type addition car la requête **whattype** retourne `+`

```
> xp:=a*x^3+b*x^2+c*x+d;#assignation du polynôme
  whattype(xp);#recherche du type principal
```

$$xp := a\,x^3 + b\,x^2 + c\,x + d$$

$$`+` \quad \textbf{(2.1.1)}$$

Une même expression peut avoir plusieurs types, un type principal et un ou plusieurs types plus élaborés. Un test sur une expression permet de savoir si elle appartient ou non à un type donné. La commande est **type(xp,nom_de_type)** où **xp** est l'expression et **nom_de_type** une catégorie d'objets se trouvant dans la liste des types reconnus par Maple. La réponse est true ou false selon que l'expression est ou non reconnue comme appartenant au type.

```
> type(xp,`+`);#xp est-il du type addition?
  type(xp,polynom);#xp est-il un polynôme?
  type(xp,anything);#xp est-il n'importe quoi?
  type(xp,ratpoly);#xp est-il un rapport de deux polynômes à
  coefficients constants?
  type(xp,matrix);#xp est-il une matrice?
```

true

true

true

true

false **(2.1.2)**

Insistons sur le fait qu'il est exclu de demander le type d'une expression invalide, c'est à dire syntaxiquement incorrecte :

```
> whattype(a*(4+s);#l'expression a*(4+s est mal formée
Error, `;` unexpected
```

Dans un registre proche, il est exclu de manipuler n'importe comment des expressions valides car leur type fixe l'étendue de leurs propriétés. Ici, on constate qu'il n'est pas possible de diviser deux chaînes de caractères (string) entre elles :

```
> a:="Erreur de saisie";whattype(a);
  b:="Recommencez";whattype(b);
  a/b;
```

a := "Erreur de saisie"

string

b := "Recommencez"

string

```
Error, invalid terms in product: Erreur de saisie
```

Types courants

On s'intéresse dans cette sous-section aux types les plus fréquemment rencontrés en programmation basique, en clair dans une feuille de travail qui fait appel aux fonctionnalités pré programmées ou à des procédures "légères" (ce qui ne veut surtout pas dire inintéressantes ...)
Les domaines couverts concernent :
• les nombres et les intervalles de nombres
• les opérations et les fonctions
• les suites, ensembles et listes
• les tableaux et les tables
• les chaînes de caractères, les noms et les symboles

Nombres

Un nombre est une expression réduite à un seul objet. Les mathématiques nous ont appris que tous les nombres n'ont pas le même statut. Il ne faut donc pas s'étonner qu'il en existe de plusieurs types.
Le classement des nombres suivant leur type a déjà été abordé dans cet ouvrage. Rappelons ici qu'ils sont classés à titre principal en entiers (integer), rationnels (rational), nombres écrits avec au moins une décimale (float), réels (realcons) et complexes (complex).

Les entiers integer sont eux mêmec classés suivant divers sous-types tels que entiers positifs (posint), entiers négatifs (negint), entiers impairs (odd), entiers pairs (even).

```
> whattype(5);
  whattype(-3);
  type(5,posint);
  type(-3,negint);
  type(5,odd);
  type(-3,even);
```

integer

integer

true

true

true

false **(2.2.1.1)**

Les nombres rationnels sont du type rational quand ils se présentent comme quotients d'un nombre entier par un nombre entier non nul. Ils regroupent les types integer et les fractions (fraction) :

```
> type(5,rational);
  type(22/7,rational);
  whattype(22/7);
```

true

true

fraction **(2.2.1.2)**

Les nombres avec partie entière et partie décimale (séparées par un point comme c'est l'usage dans le monde anglo-saxon) sont du type *float.* Ils réunissent les nombres rationnels et réels qui se présentent sous cette forme quand ils ont été évalués par la commande **evalf**.

```
> whattype(5.025);#nombre avec partie entière et partie
  décimale
  whattype(evalf(22/7));#passage du type rational au type
  float d'une fraction
  whattype(sqrt(2));
  whattype(evalf(sqrt(2)));#passage au type float d'une
  racine carrée
  whattype(Pi);
  whattype(evalf(Pi));#passage au type float d'un nombre
  symbolique
  whattype(exp(1));
  whattype(evalf(exp(1)));#passage au type float du nombre
  e=exp(1)
```

float

float

`^`

float

symbol

float

function

float **(2.2.1.3)**

Le type *realcons* réunit tous les nombres réels au sens suivant : tous les nombres exposés jusqu'ici et qui, une fois évalués par la commande **evalf,** comportent une partie entière et une partie décimale, à quoi s'ajoutent les "nombres" **infinity** et **-infinity**.

```
> type(5,realcons);
  type(-3,realcons);
  type(22/7,realcons);
```

```
type(sqrt(2),realcons);
type(Pi,realcons);
type(exp(1),realcons);
type(infinity,realcons);
type(-infinity,realcons);
```

true
true
true
true
true
true
true
true **(2.2.1.4)**

Enfin, Maple attribue le type *complex* dès qu'un nombre a une partie réelle et une partie imaginaire.

```
> type(-3+22/7*I,complex);
```

true **(2.2.1.5)**

Intervalles (range)

Un intervalle [*a*; *b*] où *a* et *b* sont des nombres s'écrit **a..b** : borne gauche suivie de deux points puis borne droite. Le résultat est du type **range** typographié par deux points (...)

```
> a:='a':b:='b':#désassignation de a et b
  ival:=a..b;
  whattype(ival);
  type(ival,range);
```

ival := *a* .. *b*

..

true **(2.2.2.1)**

Les opérandes de niveau 1 sont les deux bornes qu'on peut donc extraire isolément en cas de besoin.

```
> op(ival);
  op(1,ival);#extraction de la borne gauche
  op(2,ival);#extraction de la borne droite
```

a, *b*

a

b **(2.2.2.2)**

Opérations

Les trois opérations de base sont l'addition, la multiplication commutative et l'exponentiation, repérées respectivement par les types `+`, `*` et `^` (qui est équivalent à `**`).

L'exponentiation n'est pas forcément évidente. Si l'expression x^a est logiquement reconnue par le type `^`, il peut sembler contre-intuitif que l'expression $\frac{1}{x^a}$ le soit également, mais cela provient du fait que Maple la transforme automatiquement en expression équivalente x^{-a}, qui

est bel et bien une exponentiation.

```
> whattype(x^(-1/6));
  whattype(1/x^(1/6));#Maple refuse d'attribuer à cette
  expression le type multiplication (voir plus bas)
```

```
`^`
`^`
```

(2.2.3.1)

L'addition et la soustraction appartiennent au même type car Maple transforme automatiquement l'expression $X - Y$ en $X + (-Y)$.

```
> whattype(x^7+cos(Pi/3));
  whattype(x^7-cos(Pi/3));
  whattype(A+b-I);
```

```
`+`
`+`
`+`
```

(2.2.3.2)

La multiplication commutative et la division appartiennent au même type car Maple transforme automatiquement l'expression $\frac{X}{Y}$ en $X \cdot Y^{-1}$ quand $X \neq 1$.

```
> whattype(X*Y);
  whattype(A/B);
```

```
`*`
`*`
```

(2.2.3.3)

Il existe un type correspondant à la multiplication non commutative (par exemple, pour le produit de deux matrices) qui est symbolisé par `.` :

```
> type(M.N,`.`);
```

true **(2.2.3.4)**

Fonctions (procedure, function)

Qu'elle ait été construite par les programmeurs du logiciel ou par l'utilisateur, une fonction au sens mathématique du terme est reconnue comme du type `procedure` (le mot procedure est encadré par des accents graves).

```
> #Cas de 3 procédures pré programmées
  type(cos,`procedure`);
  type(exp,`procedure`);
  type(log,`procedure`);
  #Cas de 2 procédures créées par l'utilisateur à l'aide de
  l'opérateur flèche ->
  F:=t->cos(t)+sin(t);
  type(F,`procedure`);
  f:=(h,k,m)->alpha(h*k)^(m+6);
  type(f,`procedure`);
  #Cas d'une procédure créée par l'utilisateur à l'aide de
  proc ... end proc
  G:=proc()
  x^alpha+y^(1-alpha)
```

```
end proc;
type(G,`procedure`);
```

$$true$$

$$true$$

$$true$$

$$F := t \rightarrow \cos(t) + \sin(t)$$

$$true$$

$$f := (h, k, m) \rightarrow \alpha(h\ k)^{m+6}$$

$$true$$

$$G := \mathbf{proc}(\ )\ x\text{^}\mathrm{alpha} + y\text{^}(1 - \mathrm{alpha})\ \mathbf{end\ proc}$$

$$true$$ **(2.2.4.1)**

De façon contre intuitive, le type function est reconnu sur les images prises par une foncton mathématique élémentaire et ne s'applique que sur des expressions telles que **f(X)** où **f** est un nom et **X** une suite, éventuellement vide, d'arguments.

```
> whattype(h(x,y,z));#cas d'une fonction muette de 3
  variables
  whattype(j());#cas d'une pure fonction muette
  whattype(cos(a));#cas d'une fonction explicite
  type(cos(x),function);#l'image est reconnue comme function
  type(cos,function);#la correspondance n'est pas reconnue
  comme fonction
```

$$function$$

$$function$$

$$function$$

$$true$$

$$false$$ **(2.2.4.2)**

Si la correspondance **f** n'est pas un nom "pur", son image n'est pas du type function. Dans l'exemple suivant, la fonction-procédure **f** donne des images du type `+` puisque son évaluation est la somme des images des fonctions cosinus et sinus :

```
> f:=x->cos(x)+sin(x);
  type(f,`procedure`);
  whattype(f(theta));#Maple analyse en fait le type de
  l'expression cos(theta)+sin(theta)
```

$$f := x \rightarrow \cos(x) + \sin(x)$$

$$true$$

$$`+`$$ **(2.2.4.3)**

Le type function est reconnu en empêchant l'évaluation de **f** (par encadrement d'apostrophes) :

```
> type('f'(),function);
  type('f'(theta),function);#Maple lit f(theta)
  whattype('f'(theta));
```

$$true$$

true

function **(2.2.4.4)**

Suite (sequence)

Une suite de termes (en anglais : sequence) est une famille finie d'expressions quelconques séparées par des virgules. Le nom du type Maple est exprseq.

```
> suite1:=1,2,Pi,exp(4);#suite de nombres
  whattype(suite1);
```

$$suite1 := 1, 2, \pi, e^4$$

exprseq

```
> suite2:=x,y,z,t;#suites de variables muettes
  whattype(suite2);
```

$$suite2 := x, y, z, t$$

exprseq

```
> suite3:=cos(theta)+sin(theta),Matrix(2,2,[[a,b],[c,d]]),
  "Salut les copains",`proc() x+y^alpha end proc;`;#pot
  pourri
```

$$suite3 := \cos(\theta) + \sin(\theta), \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \text{"Salut les copains"}, \quad \textbf{(2.2.5.1)}$$

proc() x+y^alpha end proc;

Une première propriété essentielle de la sequence est qu'une fois créée, **elle garde l'ordre et la répétition éventuelle de ses termes** (ce qui ne sera pas le cas pour un ensemble). Chaque terme est donc indicié par un entier à partir de 1 et il est possible de l'extraire en précisant son rang entre crochets.

```
> suite1[1];#appel du premier terme de la sequence suite1
  suite2[4];#appel du 4° terme de la sequence suite2
  suite3[2];#appel du 2° terme de la sequence suite3
```

$$1$$

$$t$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Une seconde propriété, très utile en pratique, tient à la possibilité d'opérer des **assignements multiples** en s'assurant que le nombre de termes du membre de gauche soit celui du membre de droite :

```
> f,g,h:=3,6,1;
  f;
  h;
```

$$f, g, h := 3, 6, 1$$

$$3$$

$$1$$

Troisième propriété : on peut réunir deux sequences par une virgule.

```
> S1:=licence,master,doctorat;
```

```
S2:=maternelle,collège,lycée;
S3:=activité,chômage,retraite;
S2,S1;
S1,S3;
S2,S1,S3;
```

S1 := licence, master, doctorat

S2 := maternelle, collège, lycée

S3 := activité, chômage, retraite

maternelle, collège, lycée, licence, master, doctorat

licence, master, doctorat, activité, chômage, retraite

maternelle, collège, lycée, licence, master, doctorat, activité, chômage, retraite

Ces 3 propriétés n'épuisent pas le sujet. L'utilisation des suites est si courante qu'il faut creuser plus avant diverses possibilités offertes dans ce domaine.

L'opérateur de répétition "dollar", soit **$**, construit une suite de **n** expressions **xp** identiques avec la syntaxe **xp$n** :

```
> H(x,y,z)$6;
  whattype(%);
```

$H(x, y, z), H(x, y, z), H(x, y, z), H(x, y, z), H(x, y, z), H(x, y, z)$

exprseq **(2.2.5.2)**

Plus utile : la commande **seq** sert à créer des suites de termes récurrents. Elle sera largement commentée dans le chapitre consacré aux suites numériques. Il suffit ici d'indiquer qu'elle a pour syntaxe basique **seq(f(i),i=a..b)** où **f(i)** est une expression dépendant de la variable entière **i**, variant entre les nombres **a** et **b** ($a \leq b$) avec un pas de 1 par défaut.

```
> seq(cos(k*Pi/6),k=0..12);#création d'une table des cosinus
  remarquables
  whattype(%);#on vérifie bien que la commande seq génère
  une expression du type sequence
```

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

exprseq

Certaines commandes Maple s'applique non seulement à des objets isolés, mais aussi à des sequences. C'est en particulier le cas des deux opérateur de concaténation **cat** et **||**, qui permettent de fusionner des éléments ou des expressions. L'effet des opérateurs est le même s'ils s'appliquent sur des singletons. Par exemple, pour écrire la différentielle *dx*, on aura :

```
> dx1:=cat(d,x);
  dx2:=d||x;
  2*dx1;
  dx2^alpha;
```

dx1 := dx

dx2 := dx

$2\,dx$

dx^{α}

En revanche, l'effet est différent sur des sequences dont le nombre de termes est plus grand que 1. L'opérateur **||** opère une distribution sur les termes de la suite alors que **cat** fusionne tous les termes.

```
> S:=1,2,3,4;
  d||S;
  cat(d,S);
```

$$S := 1, 2, 3, 4$$
$$d1, d2, d3, d4$$
$$d1234$$

Ensembles (set)

Un ensemble est une suite d'objets quelconques placée entre accolades. **{ }**. Le nom du type est set.

```
> suite:=1,2,3,a*x+b,"théorie ensembliste",n..p;#création
  d'une suite
  ens:={suite};#création d'un ensemble à partir de la suite
  whattype(ens);#requête sur le type de ensemble
```

$$suite := 1, 2, 3, a\,x + b, \text{"théorie ensembliste"}, n\,..p$$
$$ens := \{1, 2, 3, \text{"théorie ensembliste"}, a\,x + b, n\,..p\}$$
$$set$$

Pour Maple, l'objet ensemble a les propriétés mathématiques des ensembles. Par conséquent, **l'ordre des éléments est indifférent et il ne peut y avoir répétition d'éléments identiques**. A noter qu'il n'est pas possible de contrôler l'ordre dans lequel Maple place les éléments dans l'ensemble. L'algorithme interne de classement est en effet aléatoire. Du coup, la même requête ne donne pas le même output d'une séance de travail à l'autre et même d'une validation à l'autre au cours de la même session ...

```
> {a,b,c};{b,a,c};{c,b,a};{a,a,a,a,b,b,b,c,b,a};#4 ensembles
  identiques en dépit des apparences
```

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

L'ensemble vide, dont on rappelle que c'est un ensemble qui n'a pas d'élément, s'écrit **{}** .

On peut former de nouveaux ensembles par réunion (commande **union**), intersection (commande **intersect**) et différence ensembliste (commande **minus**).

```
> ens1:={1,2,3,4};ens2:={3,4,5,6};#création de deux
  ensembles
  ens3:=ens1 union ens2;#réunion
  ens4:=ens1 intersect ens2;#intersection
  ens5:=ens1 minus ens2;#différence
  ens6:=ens4 intersect ens5;#exemple d'ensemble vide
```

$$ens1 := \{1, 2, 3, 4\}$$
$$ens2 := \{3, 4, 5, 6\}$$
$$ens3 := \{1, 2, 3, 4, 5, 6\}$$

$$ens4 := \{3, 4\}$$
$$ens5 := \{1, 2\}$$
$$ens6 := \{\ \}$$

On transforme un ensemble E en suite de termes par la commande **op(E)**, qui retourne les opérandes de l'ensemble E, c'est à dire les éléments qui le composent séparés par une virgule. Le nombre d'éléments de l'ensemble S est donné par **nops(E)**. L'élément i de l'ensemble est extrait par **op(i,E)**, étant entendu que $i \leq nops(E)$.

```
> op(ens1);#passage de l'ensemble ens1 à une suite de termes
  op(1,ens2);#extraction du premier élément de l'ensemble
  ens2
  nops(ens1);#nombre d'éléments de l'ensemble ens1
  nops(ens2);#nombre d'éléments de l'ensemble ens2
```

$$1, 2, 3, 4$$
$$3$$
$$4$$
$$4$$

Quand on doit exécuter des manipulations lourdes sur les éléments d'un ensemble, il peut être utile d'utiliser conjointement un ensemble et l'instruction **map** ou **map2** afin d'éviter de répéter des écritures longues.

De manière très générale, **map(proc,xp)** applique la procédure **proc** aux opérandes de l'expression **xp** et retourne une expression du même type que **xp**. Dès lors, si **xp** est un ensemble, l'output est aussi un ensemble dont les éléments ont été remplacés par leur image par la procédure. Par exemple, si on veut exprimer les valeurs prises par la fonction f en plusieurs points a, b, c et d, puis calculer les dérivées premières en chaque point, on écrira :

```
> restart;
  map(U,{a,b,c,d});#map applique la fonction d'une variable
  U à chaque élément de l'ensemble
  map(D(U),{a,b,c,d});#map applique la procédure de
  dérivation de U à chaque élément de l'ensemble
```

$$\{U(a), U(b), U(c), U(d)\}$$
$$\{\mathrm{D}(U)(a), \mathrm{D}(U)(b), \mathrm{D}(U)(c), \mathrm{D}(U)(d)\} \qquad \textbf{(2.2.6.1)}$$

La syntaxe basique de **map2** demande 3 arguments qui sont, dans l'ordre :

1. une procédure **proc** qui a elle-même deux arguments obligatoires,
2. un argument **arg** qui est le premier argument exigé par la procédure **proc**
3. une expression **xp** dont les éléments constituent chacun le deuxième argument de la procédure.

La commande **map2(proc,arg,xpr)** retourne une expression du même type que **xp**. Si **xp** est un ensemble, l'output est un nouvel ensemble qui a remplacé chaque élément de départ par leur image.

Par exemple, pour former un ensemble regroupant les dérivées partielles premières d'une fonction de 4 variables, on devra écrire in extenso :

```
> F:=(x,y,z,t)->exp(-(x^2+y^2+z^2+t^2)/2);#définition d'une
  fonction de trois variables
```

```
{diff(F(x,y,z,t),x),diff(F(x,y,z,t),y),diff(F(x,y,z,t),z),
diff(F(x,y,z,t),t)};#ensemble formé des dérivées
partielles premières
```

$$F := (x, y, z, t) \rightarrow e^{-\frac{1}{2}x^2-\frac{1}{2}y^2-\frac{1}{2}z^2-\frac{1}{2}t^2}$$

$$\left\{-t\,e^{-\frac{1}{2}x^2-\frac{1}{2}y^2-\frac{1}{2}z^2-\frac{1}{2}t^2}, -x\,e^{-\frac{1}{2}x^2-\frac{1}{2}y^2-\frac{1}{2}z^2-\frac{1}{2}t^2}, -y\,e^{-\frac{1}{2}x^2-\frac{1}{2}y^2-\frac{1}{2}z^2-\frac{1}{2}t^2}, -z\,e^{-\frac{1}{2}x^2-\frac{1}{2}y^2-\frac{1}{2}z^2-\frac{1}{2}t^2}\right\} \quad \textbf{(2.2.6.2)}$$

Mais il est plus rapide d'invoquer **map2** :

```
> map2(diff,F(x,y,z,t),{x,y,z,t});
```

$$\left\{-t\,e^{-\frac{1}{2}x^2-\frac{1}{2}y^2-\frac{1}{2}z^2-\frac{1}{2}t^2}, -x\,e^{-\frac{1}{2}x^2-\frac{1}{2}y^2-\frac{1}{2}z^2-\frac{1}{2}t^2}, -y\,e^{-\frac{1}{2}x^2-\frac{1}{2}y^2-\frac{1}{2}z^2-\frac{1}{2}t^2}, -z\,e^{-\frac{1}{2}x^2-\frac{1}{2}y^2-\frac{1}{2}z^2-\frac{1}{2}t^2}\right\}$$

Malheureusement, l'ordre dans lequel on aimerait voir apparaître les dérivées partielles n'est pas respecté. C'est logique puisque l'ordre ne compte pas dans un ensemble et qu'on ignore tout de la manière dont Maple fonctionne "en interne". Ce désagrément n'existe pas quand on manipule des listes.

Listes (list)

Formellement, une liste (list) est une suite de termes quelconques mise entre crochets [ ].

```
> liste1:=[1,Pi,sqrt(2),exp(4),"matrice"];#écriture directe
  d'une liste
  whattype(liste1);
  suite:=seq(x^i,i=1..7);#création d'une suite de termes
  liste2:=[suite];#passage de la suite à une liste
  whattype(liste2);
```

$$liste1 := [1, \pi, \sqrt{2}, e^4, \text{"matrice"}]$$

$$list$$

$$suite := x, x^2, x^3, x^4, x^5, x^6, x^7$$

$$liste2 := [x, x^2, x^3, x^4, x^5, x^6, x^7]$$

$$list$$

Une liste est une famille de termes ordonnés et chacun de ces termes se voit attribuer un indice, c'est à dire un entier positif indiquant sa place dans la liste. En conséquence, une liste, contrairement à un ensemble, respecte l'ordre dans lequel figurent ses éléments, ainsi que la répétition éventuelle des termes :

```
> [a,b,c];[a,c,b];[c,a,a,a,b];[a,c,a,b,a];
```

$$[a, b, c]$$

$$[a, c, b]$$

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

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

Ces deux propriétés rendent une liste plus intéressante et attractive qu'un ensemble. Pour

transformer un résultat présenté dans un ensemble **ensemble** en résultat sous forme de liste, on applique la commande de conversion **convert(ensemble,list)**.

```
> ensemble:={a,b,"a",b,c,d,e};
  convert(ensemble,list);
```

$$ensemble := \{\text{"a"}, a, b, c, d, e\}$$

$$[\text{"a"}, a, b, c, d, e] \qquad \textbf{(2.2.7.1)}$$

Réciproquement, on passe d'une liste **liste** à un ensemble par la commande **convert(liste,set)**. On perd alors les propriétés d'ordre et de répétition :

```
> convert(liste1,set);
  convert([a,c,a,b,a],set);
```

$$\{1, \text{"matrice"}, \pi, \sqrt{2}, e^4\}$$

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

Les éléments d'une liste **liste** sont ses opérandes de niveau 1. Leur nombre est donné par **nops(liste)** et on passe d'une liste à une sequence avec **op(liste)**.

```
> nops(liste1);nops(liste2);
  op(liste1);op(liste2);
```

$$5$$

$$7$$

$$1, \pi, \sqrt{2}, e^4, \text{"matrice"}$$

$$x, x^2, x^3, x^4, x^5, x^6, x^7 \qquad \textbf{(2.2.7.2)}$$

L'extraction du *i*-ème élément se fait par **op(i,liste)** où **i** est un entier compris entre 1 et **nops(liste)** :

```
> op(2,liste1);op(3,liste1);op(3,liste2);
```

$$\pi$$

$$\sqrt{2}$$

$$x^3 \qquad \textbf{(2.2.7.3)}$$

Mais on extrait plus simplement un terme d'une liste en l'appelant par son numéro d'ordre encadré par deux crochets. Le principe est le même pour former une sous-liste : la plage d'indices est signalée par un intervalle placé entre crochets.

```
> liste1[2];liste1[3];liste2[3];#extraction d'un élément
  liste2[3..6];#extraction d'une sous-liste
```

$$\pi$$

$$\sqrt{2}$$

$$x^3$$

$$[x^3, x^4, x^5, x^6]$$

Pour remplacer le *i*-ème terme **i** d'une liste donnée **liste** par une autre expression **xpr**, on utilise la commande **subsop(i=xpr,liste)**. Comme Maple ne modifie pas l'assignation antérieure, il faut assigner le résultat à une nouvelle liste :

```
> subsop(4=exp(alpha)+exp(beta),liste2);
```

```
liste2;#liste2 est inchangée.
liste2a:=subsop(4=exp(alpha)+exp(beta),liste2);#la
modification du 4° terme est prise en compte dans la
nouvelle liste.
```

$$[x, x^2, x^3, e^{\alpha}+e^{\beta}, x^5, x^6, x^7]$$

$$[x, x^2, x^3, x^4, x^5, x^6, x^7]$$

$$liste2a := [x, x^2, x^3, e^{\alpha}+e^{\beta}, x^5, x^6, x^7]$$

La concaténation de deux ou plusieurs listes consiste à mettre entre crochets les opérandes des listes concernées. Ainsi, on passe d'une liste à une sequence par la commande **op**.

```
> liste3:=[op(liste1),op(liste2),liste2a[4]];
```

$$liste3 := [1, \pi, \sqrt{2}, e^4, \text{"matrice"}, x, x^2, x^3, x^4, x^5, x^6, x^7, e^{\alpha}+e^{\beta}]$$

Tableaux (Array)

Le type tableau (en français)/Array (en anglais) généralise le type liste/list dans la mesure où il autorise une indiciation numérique multiple de termes quelconques. A titre d'analogie, un élément d'un vecteur est repéré par un indice, qui est son numéro d'ordre; un élément d'une matrice est repéré par deux indices, le premier indiquant la ligne, le second la colonne où il se trouve. Rien n'interdit d'imaginer qu'un élément soit repéré par trois nombres entiers positifs. Avec un peu d'audace, on pourrait suggérer que les indices prennent des valeurs entières positives, nulles ou négatives. Enfin, rien n'interdit dans certaines utilisations qu'un élément soit ordonné suivant N indices (entiers positifs ou négatifs). Le type Array correspond à cette vision. Attention! Le type Array diffère du type array (avec un "a" minuscule) qui est une ancienne version maintenant dépréciée.

Tableau à une dimension

Dans le cadre défini ci-dessus, une liste s'apparente à un tableau à une dimension :

```
> restart;
  liste:=[1,2,3,4];#création d'une liste
  tableau:=Array(1..4,[1,2,3,4]);#création d'un tableau
  whattype(liste);
  whattype(tableau);
```

$$liste := [1, 2, 3, 4]$$

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

list

Array

On voit bien qu'en dépit des apparences, le type n'est pas le même.
Une différence essentielle est que c'est l'utilisateur qui fixe l'intervalle des indices. La commande **`Array(m..n)`** crée un tableau uni-ligne dont le premier élément est indicié par *m* (*m* entier), le second par (*m*+1), .., le dernier par *n* (*n* entier strictement supérieur à *m*) . La commande **`Array(m..n,[suite de (n-m+1) termes])`** crée un tableau uni-ligne dont les éléments sont ceux de la liste à $(n-m+1)$ termes et sont indiciés de *m* à *n*. On extrait un élément du tableau en l'appelant par son numéro d'ordre dans l'intervalle *m..n*.

```
> restart;
  tableau:=Array(-2..1,[1,2,3,4]);
```

```
  tableau[-1];tableau[0];tableau[2];
```

$$tableau := Array(-2..1, \{-2 = 1, -1 = 2, 0 = 3, 1 = 4\}, datatype = anything, storage = rectangular, order = Fortran\_order)$$

$$2$$

$$3$$

```
Error, Array index out of range
```

En pratique, il peut être utile de commencer par déclarer la dimension du tableau :

```
> carres:=Array(1..4);
```

$$carres := \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}$$

Puis on assigne les éléments du tableau :

```
> carres[1]:=0;carres[2]:=1;carres[3]:=4;carres[4]:=9;
```

$$carres_1 := 0$$

$$carres_2 := 1$$

$$carres_3 := 4$$

$$carres_4 := 9$$

Pour afficher le contenu du tableau, il suffit d'appeler son nom :

```
> carres;
```

$$\begin{bmatrix} 0 & 1 & 4 & 9 \end{bmatrix} \qquad \textbf{(2.2.8.1.1)}$$

Tableaux à deux dimensions

La commande **`Array(m..n,p..q)`** déclare un tableau indicié où le premier indice varie de m à n et le second indice de p à q. On peut alors remplir le tableau en écrivant les $(n-m+1)(q-p+1)$ éléments. Il est aussi possible de remplir directement le tableau par la commande **`Array(m..n,p..q,[[elmp,..,elmq],..,[elnp,..,elnq]])`**.

```
> restart;
  tableau:=Array(-1..1,4..6);
  tableau[-1,4]:=1;tableau[-1,5]:=1;tableau[-1,6]:=1;
  tableau[0,4]:=2;tableau[0,5]:=4;tableau[0,6]:=8;tableau
  [1,4]:=3;tableau[1,5]:=9;tableau[1,6]:=27;
  tableau;#affichage du contenu
```

$$tableau := Array(-1..1, 4..6, \{\}, datatype = anything, storage = rectangular, order = Fortran\_order)$$

$$tableau_{-1,4} := 1$$

$$tableau_{-1,5} := 1$$

$$tableau_{-1,6} := 1$$

$$tableau_{0,4} := 2$$

$$tableau_{0,5} := 4$$

$$tableau_{0,6} := 8$$

$$tableau_{1,4} := 3$$

$$tableau_{1,\,5} := 9$$

$$tableau_{1,\,6} := 27$$

$$Array(-1..1, 4..6, \{(-1, 4) = 1, (-1, 5) = 1, (-1, 6) = 1, (0, 4) = 2, (0, 5) = 4, (0, 6) = 8, (1, 4) = 3, (1, 5) = 9, (1, 6) = 27\}, datatype = anything, storage = rectangular, order = Fortran\_order)$$

```
> restart;
  Array(-1..1,4..6,[[1,1,1],[2,4,8],[3,9,27]]);
```

$$Array(-1..1, 4..6, \{(-1, 4) = 1, (-1, 5) = 1, (-1, 6) = 1, (0, 4) = 2, (0, 5) = 4, (0, 6) = 8, (1, 4) = 3, (1, 5) = 9, (1, 6) = 27\}, datatype = anything, storage = rectangular, order = Fortran\_order)$$

Moyennant les aménagements nécessaires, toutes les remarques faites dans la présentation des tableaux uni-lignes restent valables en deux dimensions.

Tableaux à plus de deux dimensions

Les objets de type Array ne sont pas limités à une ou deux dimensions, et il est parfaitement possible de construire des tableau à trois dimensions ou plus (le maximum étant 63 dimensions!) ; par exemple :

```
> restart;tableau:=Array(1..2,1..2,1..2,[[[a111,a112],
  [a121,a122]],[[a211,a212],[a221,a222]]]);
```

$$tableau := \begin{bmatrix} 1..2 \times 1..2 \times 1..2\ Array \\ Data\ Type: anything \\ Storage: rectangular \\ Order: Fortran\_order \end{bmatrix}$$

Evidemment, il n'est pas possible de visualiser un tel tableau mais, en "double-cliquant gauche" sur l'output, on accède à son contenu par une fenêtre de navigation. On peut alors exporter des informations en une ou deux dimensions.

Tables (tables)

Les tables/table généralisent la notion de tableau/Array au sens où elles admettent n'importe quel système d'indiçage (nombres, noms ou même formules). Un petit exemple suffira pour faire comprendre le principe :

```
> lille1:=table([sc_eco=["2 départements","2000 étudiants"],
  MASS=["1 département","150 étudiants"],maths=["1
  département","400 étudiants"]]);
  type(lille1,table);
```

$$lille1 := table([maths = ["1\ département", "400\ étudiants"], MASS = ["1\ département", "150\ étudiants"], sc\_eco = ["2\ départements", "2000\ étudiants"]])$$

$$true$$

```
> lille1[sc_eco];lille1[MASS];lille1[maths];
```

["2 départements", "2000 étudiants"]

["1 département", "150 étudiants"]

["1 département", "400 étudiants"]

Chaînes de caractères (string)

Une chaîne de caractères est une succession de ... caractères alphabétiques (y compris l'espacement) ou numériques placés entre guillemets. La validation de l'input affiche la chaîne encadrée par des guillemets.

```
> "Le type string est une suite de caractères placée entre
  guillemets ";
  whattype(%);
```

"Le type string est une suite de caractères placée entre guillemets "

string

Elle peut être assignée mais elle ne peut servir à assigner.

```
> Ch:="Le type string est une suite de caractères placée
  entre guillemets ";
  Ch;
  "this":=1;
```

Ch := "Le type string est une suite de caractères placée entre guillemets "

"Le type string est une suite de caractères placée entre guillemets "

```
Error, invalid left hand side of assignment
```

A l'instar d'une liste, on peut extraire des éléments d'une chaîne de caractères parce que ces derniers sont indiciés, y compris les espaces :

```
> Ch[2];Ch[3];Ch[1..14];
```

"e"

" "

"Le type string"

Le nombre d'éléments d'une chaîne de caractères **Ch** est donné par la commande **length(Ch)** :

```
> length(Ch);
```

67

On réunit deux chaînes par concaténation avec **cat:**

```
> Ch1:="à boire";Ch2:="et";Ch3:="à manger";
  cat(Ch1," ",Ch2," ",Ch3);
  cat(Ch3," ",Ch2," ",Ch1);
```

Ch1 := "à boire"

Ch2 := "et"

Ch3 := "à manger"

"à boire et à manger"

"à manger et à boire"

Noms (name)

A priori, un nom est une chaîne de caractères. Ce n'est pas le cas en informatique où un nom sert le plus souvent à donner une adresse à une expression, opération d'assignation interdite pour les chaînes de caractères (voir ci-dessus).

Par définition, un objet du type name est formé de un ou plusieurs caractères non espacés. Un nom à un caractère est nécessairement formé par une lettre majuscule ou minuscule. Un nom à plusieurs caractères commence indifféremment par une lettre majuscule ou minuscule, un

chiffre différent de zéro ou un caractère spécial tel que _ (tiret bas) ou ~ (tilde).

```
> #exemples de noms à 1 caractère
  type(F,name);
  type(f,name);
  type(8,name);#le caractère est un chiffre
  #exemples de noms à plus de 1 caractère
  type(Macroéconomie1,name);
  type(1Macroéconomie,name);#le nom commence par un chiffre
  type(_Macroéconomie,name);#le nom commence par un tilde
  type(0Mac,name);#un message d'erreur signale que le nom
  commence par un zéro
                                    true
                                    true
                                    false
                                    true
                                    true
                                    true
Error, missing operator or `;`
```

Les limitations signalées disparaissent si le nom est encadré par des accents graves (obtenus chacun par la combinaison de touches Ctrl+Alt+le 7 du pavé alphabétique, suivie de Enter).

```
> #exemples de noms à 1 caractère
  type(`8`,name);
  type(`_`,name);
  type(`^`,name);
  #exemple de nom à plus de 1 caractère
  type(`0Mac`,name);
                                    true
                                    true
                                    true
                                    true                    (2.2.11.1)
```

Le type name est subdivisé en deux sous-types : symbol et indexed. Le type symbol regroupe les noms ordinaires comme ceux qu'on vient de former.

```
> whattype(F);
  whattype(f);
  whattype(1Macroéconomie);
  whattype(`0Mac`);
                                    symbol
                                    symbol
                                    symbol
                                    symbol                  (2.2.11.2)
```

Le type **indexed** s'applique aux noms finissant par une liste (généralement, une liste de chiffres pour faciliter le classement des noms).

```
> whattype(Macro[1]);
```

```
whattype(Macro[2]);
```

indexed

indexed **(2.2.11.3)**

Bien que rarement pratiquée, l'assignation au moyen de noms indiciés est licite :

```
> Macro[1]:=x->H*x^alpha;#assignation d'une procédure à un
  nom indicié
  Macro[1](2);#calcul de la valeur prise par la procédure en
  x=2
  D(Macro[1]);#calcul de la dérivée première de la fonctio-
  procédure
```

$$Macro_1 := x \rightarrow H\,x^{\alpha}$$

$$H\,2^{\alpha}$$

$$x \rightarrow \frac{H\,x^{\alpha}\,\alpha}{x} \qquad \textbf{(2.2.11.4)}$$