I. Quelques types de données Maple

1. Objectifs

L'objectif de ce document est d'introduire un certain nombre de *types* Maple qui sont proches de notions mathématiques et utiles en calcul formel. On introduira aussi des fonctions permettant de manipuler des objets (données, data) de ces types. Ces notions seront mises en œuvre à travers des résolutions d'équations et des calculs avec des *suites de Farey*.
On rassemble dans deux tableaux ces listes et ces fonctions.

types :	séquence	liste	ensemble	somme	produit
	`exprseq`	`list`	`set`	`'+'`	`'*'`

	renvoie le type	`whattype`
	renvoie la séquence d'opérandes	`op`
	renvoie le nombre d'opérandes	`nops`
	convertit	`convert`
	substitue	`subs`
fonctions :	développe	`expand`
	range suivant un ordre	`sort`
	rassemble	`collect`
	résoud	`solve`
	dérive	`diff`
	résoud une équa diff	`dsolve`

On s'attachera aussi à quelques *opérateurs* et *constructeurs*.

opérateurs	,	=	+
type résultat	séquence	égalité	somme

constructeurs	`[ ]`	`{ }`	`seq()`	`add()`
type résultat	liste	ensemble	séquence	somme

Il faut aussi avoir bien compris le principe de *l'évaluation maximale* même si ce principe admet quelques exceptions.
Pour obtenir l'aide complète, insérer un de ces mots dans une feuille Maple, placer le curseur à l'intérieur et taper F1 ou F2 suivant les versions de Maple.
Les tableaux sont introduits dans un autre paragraphe. Ce sont des types d'objets très importants dans tous les types de langage. Ils ne sont pas spécifiquement formels.

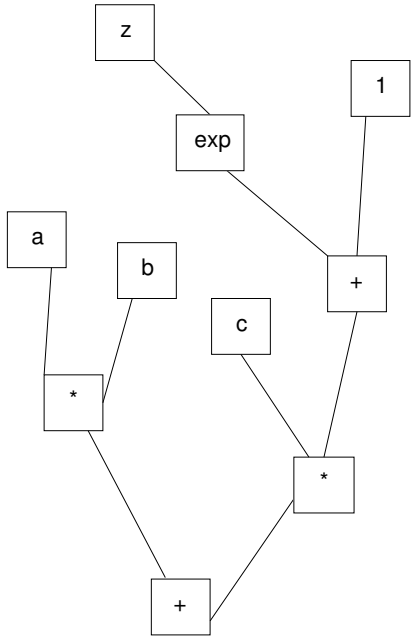

FIG. 1 – Arbre a*b+c*(exp(z)+1)

2. Expressions - Arbres - Évaluation maximale

1. Arbres

Les *expressions mathématiques formelles* sont gérées par Maple comme des *arbres* (voir figure 1). Les nœuds sont des mots (au sens habituel c'est à dire un assemblage de lettres). Il existe deux types de mots : ceux que Maple connait (on dira de tels mots qu'ils sont réservés) et les autres.
Les mots réservés sont des éléments de la syntaxe comme `while` ou le nom d'objets prédéfinis par Maple et désignant des objets mathématiques (`Pi`, `exp`) ou des commandes. Il est évidemment interdit d'assigner un objet à un mot réservé, cela influerait sur le fonctionnement même de Maple. Par exemple

```
> Pi := 5;
 Error, attempting to assign to ‘Pi‘ which is protected
```

Les autres mots (par exemple `truc`) sont considérés systématiquement comme des noms (ie des variables). Si une commande d'assignation a été exécutée, on dira

Cette création est mise à disposition selon le Contrat
Paternité-Pas d'utilisations commerciale-Partage des Conditions Initiales à l'Identique 2.0 France
disponible en ligne http://creativecommons.org/licenses/by-nc-sa/2.0/fr/

que la variable est assignée, sinon on dira qu'elle est non assignée (ou libre). Par exemple, `truc` est assignée si une commande du type

```
truc := machin;
```

a été exécutée. On dira que `truc` est assignée même si `machin` ne l'est pas.

2. Évaluation maximale

En général les noms de variables sont *évalués* par Maple. Par exemple, si `truc` est un nom assigné qui désigne un certain arbre, lorsque Maple traite ce mot, il va chercher à remplacer systématiquement (ou presque) les noms assignés par ce qu'ils désignent.

```
>restart;
 a:=2;
 truc:=a+b;
 truc;
 a:=3; truc;
>restart;
 truc:=a+b; truc;
 a:=2;truc;
 a:=3;truc;
```

De plus, Maple remplacera un certain nombre d'expression "évidentes" du genre `1+2` (mais pas `1=2`). Ce processus est appellé le *principe d'évaluation maximale*. Il est à noter que certains objets dérogent à cette règle, en particulier les tableaux et les fonctions. Ce point n'est pas abordé ici.
Pour contourner ce comportement déroutant, il est conseillé d'utiliser des substitutions.

3. Type

Le mot "racine" détermine le type de l'arbre. La fonction[1] `whattype` permet de le connaitre.

```
> whattype(1=2); whattype([truc,machin]);
> whattype({truc,machin});whattype(truc,machin);
> op(truc+machin);
> nops(a*b);
> convert(a*b,'+');
```

[1] On peut remarquer que toutes les fonctions citées ici *renvoient un résultat sans modifier leur argument*. Le paramètre est passé par valeur.

La fonction `op` permet « d'enlever »la racine et d'accéder à la séquence des noeuds du premier niveau (les *opérandes*). La fonction `nops` renvoie le nombre d'opérandes. On peut accéder aux éléments d'une liste (mais pas d'une séquence) en utilisant des crochets.

```
> truc:=[a,b,c,d]; truc[1]; truc[3]; truc[15];
```

On comprendra mieux les types introduits ici, les fonctions associées et les substitutions avec des exemples plus significatifs.

3. Substitution - Résolution d'équations

1. Résolution d'équations

Dans le code suivant, `eq1` et `eq2` sont assignés à des égalités, `{eq1,eq2}`, `{x,y}` sont évalués à des ensembles (respectivement d'équations et d'inconnues)

```
> eq1 := a*x+b*y = u;
  eq2 := c*x+d*y = v;
  solve({eq1,eq2},{x,y})
```

La fonction `solve` renvoie un ensemble d'égalités[2]. Pour une seule équation et une seule inconnue on peut se passer des ensembles `solve(a*x+b=0,x);`. La fonction ne renvoie pas alors un ensemble. Je conseille toutefois de toujours mettre les accolades car le renvoi d'un ensemble d'égalités est plus commode pour combiner ensuite avec des substitutions.

2. Substitution

Dans une expression, on peut substituer quelque chose à des noms libres.

```
> restart;
  truc := a+b;
  subs({a=machin,b=2},truc); a, b, machin, truc;
```

Un bon exemple d'utilisation est fourni par le calcul d'un projeté orthogonal sur une droite.

```
>restart;
 eqD := a*x+b*y=c; # équation de la droite
 M:= [u,v] ; # coordonnées du point
```

[2] je n'aime pas que les noms des inconnues soient réutilisés mais bon ...

Cette création est mise à disposition selon le Contrat
Paternité-Pas d'utilisations commerciale-Partage des Conditions Initiales à l'Identique 2.0 France
disponible en ligne http://creativecommons.org/licenses/by-nc-sa/2.0/fr/

```
P := [u+a*lambda, v+b*\lambda]; # pt de droite orth  par M
eq := subs({x=P[1],y=P[2]},eqD);# eq en lambda pour intersect
truc := solve({eq},{lambda});# un ensemble contenant une égalité
subs(truc,P); #les coordonnées du projeté
```

3. Résolution d'une équation différentielle

Dans le code suivant remarquer l'utilisation de `diff` pour dériver l'expression `y(x)`. La présence de parenthèses indique à Maple de considérer `y` comme une fonction. Comme `y` est la seule fonction dans l'expression, il n'est pas utile d'indiquer la fonction inconnue.

```
> eq:=diff(y(x),x,x)+(omega^2)*y(x)=0;
  dsolve(eq);
```

Prévoir ce que renverront les instructions suivantes.

```
> restart;eq:=diff(y(x),x)+a(x)*y(x)=0;
  dsolve(eq);
> restart;eq:=diff(y(x),x)+diff(a(x),x)*y(x)=0;
  dsolve(eq);
> restart;eq:=diff(y(x),x)+diff(a(x),x)*y(x)=0;
  dsolve(eq,y(x));
```

4. Suites de Farey

Pour un entier naturel n (au moins 2), la suite de Farey[3] d'ordre n est formée par les rationnels entre 0 et 1 et qui s'écrivent avec un dénominateur plus petit que n.
Ordonner à la main les éléments d'une suite de Farey est plus que fastidieux. Les éléments ordonnés d'une suite de Farey vérifient une propriété curieuse : après réduction au même dénominateur et simplification, le numérateur de la différence entre deux éléments consécutifs est toujours 1.

La fonction seq permet de construire une sequence avec une syntaxe assez naturelle[4]. Considérons par exemple la séquence des inverses des entiers de 1 à 10.

```
>seq(1/j,j=1..10);
>restart;
```

[3] John Farey (1766-1826) était un géologiste anglais dont le nom est resté attaché aux mathématiques à cause de cette propriété.

[4] elle ressemble à celle utilisée en mathématiques pour une somme. Par exemple $\sum_{j\in\{1\cdots10\}}\frac{1}{j}$

```
n:=5;
FareySeq:=seq(seq(i/j,i=1..j),j=1..n);
nops(FareySeq);
```

Cela signifie que pour Maple la sequence n'est pas vraiment un type, une sequence n'est pas un objet unique.
Une liste (`list`) est une séquence entourée par des crochets :

```
> FareyList:=[FareySeq];
```

La fonction `sort` trie les éléments d'une liste (et d'autres types de données, voir l'aide) mais pas ceux d'une séquence.

```
>FareyList:=sort(FareyList);FareySeq:=sort(FareySeq);
```

Le type ensemble (`set`) rend bien compte d'un ensemble au sens mathématique. On forme un ensemble en entourant une séquence par des accolades.

```
>FareySet:={FareySeq};
```

Un ensemble Maple contient chaque élément une seule fois (comme en maths), il est stocké et affiché selon un ordre interne à Maple et qui n'a rien de naturel.
En ce qui concerne les suites de Farey, l'intérêt est bien cette "élimination de doublons".
L'utilisation de la fonction `op` et des constructeurs `[]` et `{}` permet de passer d'un type à l'autre.

```
>FareyList:=[op(FareySet)];
```

Écrire les quelques lignes de code qui renvoient la liste ordonnée des élements d'une suite de Farey d'ordre n. Combien la suite ordonnée de Farey pour $n = 245$ contient-elle d'éléments ? Former la liste des différences entre deux termes consécutifs.

Cette création est mise à disposition selon le Contrat
Paternité-Pas d'utilisations commerciale-Partage des Conditions Initiales à l'Identique 2.0 France
disponible en ligne http://creativecommons.org/licenses/by-nc-sa/2.0/fr/