5 - Opérateurs

1. Types d'opérateurs

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

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

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

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

2. Opérateurs arithmétiques

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

Dans la catégorie *unaire*, on trouve :

`+` : préfixé, mais sans effet. Il disparait à l'affichage.

`-` : préfixé. C'est le changement de signe. Ne pas le confondre avec la soustraction.

`!` : postfixé. C'est la factorielle. Par exemple 3! vaut $3 * 2 = 6$ et 3!! vaut $6! = 720$.
L'argument de la factorielle ne doit pas être négatif.

Dans la catégorie *binaire*, on trouve :

`+` : c'est l'addition, toujours commutative et associative.
On peut considérer des sommes $a + b + c + \cdots$ sans risque d'ambiguités.
Quand on évalue une somme, les termes semblables sont automatiquement regroupés.
Si le résultat final est encore une somme, on ne peut pas prévoir l'ordre des termes.

`-` : c'est la soustraction.
En interne, $a - b$ est codé $a + (-b)$ (somme de a et de l'opposé de b).

`*` : c'est le produit, toujours commutatif et associatif.
On peut donc considérer des produits $a * b * c * \cdots$ sans risque d'ambiguités.
Quand on évalue un produit, les termes semblables sont automatiquement regroupés.
Si le résultat final est encore un produit, on ne peut pas prévoir l'ordre des termes.

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

`^` : c'est l'exponentiation. On peut utiliser le synonyme $**$.

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

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

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

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

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

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

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

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

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

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

3. Opérateurs logiques

Maple connait trois constantes symboliques logiques :

true représente la valeur logique *vrai*.

false représente la valeur logique *faux*.

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

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

Maple utilise donc une logique *trivaluée*.

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

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

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

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

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

FAIL, *false*, *FAIL*

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

true, *FAIL*, *FAIL*, *FAIL*