

Calcul formel (Maple)

TP5 : Programmation

But du TP5

Maple est un langage de calcul formel, mais c'est aussi un langage de programmation. Nous allons voir comment créer des programmes avec Maple. Pour cela, nous allons utiliser des structures communes à la plupart des langages de programmation : procédures, tests (*if*), boucles (*for*, *while*), fonctions récursives.

1. Les booléens

On appelle *expression booléenne* une expression dont l'évaluation conduit ou bien à la valeur *true* (vrai) ou bien à la valeur *false* (faux). L'évaluation des expressions booléennes se fait par la commande *evalb*.

Les expressions booléennes sont très utiles en programmation car elles permettent d'effectuer des tests qui détermineront la suite des instructions à effectuer. Voici différentes méthodes pour fabriquer des expressions booléennes..

1.1 Les opérateurs de comparaison

On peut former une expression booléenne en comparant deux expressions de même type à l'aide d'un des opérateurs suivants :

=	(égale)
<>	(est différent de)
<	(est strictement inférieur à)
>	(est strictement supérieur à)
<=	(est inférieur ou égal à)
>=	(est supérieur ou égal à)

Dans l'exemple suivant, on définit une expression booléenne avant de l'évaluer.

```
> restart;  
bool:=1<2;  
  
bool := 1 < 2 (1)
```

```
> evalb(bool);  
  
true (2)
```

```
> evalb(Pi=3.14);  
  
false (3)
```

Parfois, quand Maple ne dispose pas d'informations suffisantes, il ne peut pas évaluer l'expression booléenne ; par exemple :

```
> evalb(x>=y);  
  
y ≤ x (4)
```

Attention, la fonction *evalb* ne sait pas faire des calculs algébriques (contrairement à *simplify*), comme le montre l'exemple suivant :

```
> evalb( x^2-y^2 = (x-y) * (x+y) );  
false (5)
```

1.2 Les opérateurs logiques

Ils permettent de modifier des expressions booléennes :

not (non)
and (et)
or (ou inclusif)

(remarque : ils ne nécessitent pas d'utiliser *evalb*)

```
> not(1<2);  
false (6)
```

```
> 1<2 and 4<3;  
false (7)
```

1.3 Les fonctions booléennes

Les fonctions booléennes sont des commandes de Maple qui renvoient *true* ou *false*. Par exemple la fonction *iscont* teste la continuité d'une fonction sur un intervalle :

```
> ?iscont; iscont(tan(x),x=0..1);  
true (8)
```

Il y a aussi la fonction *type*, qui permet de tester si une expression est d'un type donné. Par exemple, la commande suivante permet de tester si Pi est un entier (*integer*) :

```
> type(Pi,integer);  
false (9)
```

(pour en savoir plus sur la commande *type* et les différents types possibles, consultez l'aide)

2. Les procédures

Une *procédure* est un petit programme qui a un nom, des entrées et une sortie. Au coeur du programme, il y a une suite d'instructions à accomplir. Une procédure utilise des variables qui lui sont propres (variables locales). Par exemple, la procédure suivante, qui se nomme *différence*, prend en entrées deux nombres *x* et *y* et renvoie leur différence.

```
> différence:=proc(x,y)  
RETURN(x-y);  
end proc;  
différence:=proc(x,y) RETURN(x - y) end proc (10)
```

```
> différence(4,5);  
-1 (11)
```

(pour aller à la ligne lorsque vous tapez une procédure, utilisez les touches Maj Entrée)

* *proc* signifie que l'on est en train de définir une procédure (attention, il ne faut pas mettre de point-virgule après *proc*).

* *RETURN* (en majuscules !) affiche le résultat de la procédure.

* *end proc* signifie que la définition de la procédure est terminée.

Voici une procédure qui se nomme *sommeproduit*, qui prend en entrées trois nombres *a*, *b* et *c*, et qui renvoie la liste formée de leur somme et leur produit. Au cours de la procédure, on stocke les calculs

intermédiaires dans des **variables locales**, qui sont auparavant déclarées par la ligne *local*.

```
> sommeproduit:=proc(a,b,c)
  local s,p;
  s:=a+b+c;
  p:=a*b*c;
  RETURN([s,p]);
end proc;
```

```
sommeproduit := proc(a, b, c)
```

```
  local s, p; s := a + b + c; p := a * b * c; RETURN([s, p])
```

```
end proc
```

(12)

```
> sommeproduit(2,3,4);
```

```
[9, 24]
```

(13)

Recommandation

En programmation, il est important de bien réfléchir à ce que l'on veut faire avant de passer sur la machine. Avec un papier et un crayon, réfléchissez d'abord au programme : ses entrées, ses instructions, ses sorties. Ensuite, implémentez-le sur Maple. Cela permet de mieux distinguer les erreurs qui viennent de la conception du programme de celles dues à une mauvaise syntaxe des commandes.

3. Le test (if)

La structure *if* permet de tester si une condition est vérifiée. Elle peut être employée dans une procédure. Elle commence par un *if* et se termine par un *end if*. Voici un exemple avec une procédure qui calcule le minimum de deux nombres :

```
> minimum:=proc(x,y)
  if x<y then RETURN(x) else RETURN(y) end if ;
end;
```

```
> minimum(7,-2);
```

```
-2
```

(14)

Les syntaxes possibles sont les suivantes. A chaque fois, les conditions sont des expressions booléennes et les instructions sont des commandes Maple.

* Pour une exécution conditionnelle :

```
if Condition then instructions end if;
```

Maple évalue d'abord l'expression booléenne "Condition". Si le résultat est *true*, alors les instructions sont effectuées. Sinon, Maple passe à la suite (après le *end if*).

* Pour un choix binaire :

```
if Condition then instructions_1 else instructions_2 end if;
```

Maple évalue d'abord l'expression booléenne "Condition". Si le résultat est *true*, alors les "instructions_1" sont effectuées. Sinon, Maple effectue les "instructions_2". Sinon, Maple passe à la suite (après le *end if*).

* Pour un choix multiple :

```
if condition_1 then instructions_1
```

```
elif condition_2 then instructions_2
```

```
...
```

```
elif condition_n then instructions_n
```

```
else instructions_(n+1)
```

```
end if
```

Maple évalue l'expression booléenne `condition_1`. Si le résultat est *true*, alors les instructions `_1` sont effectuées et Maple passe à la suite (après le ***end if***). Si c'est *false*, Maple évalue l'expression booléenne `condition_2` et procède de même. Si aucune des expressions booléennes `condition_1, ..., condition_n` n'est vraie, Maple effectue les instructions `_(n+1)`.

(rappels pour les non-anglophones : if=si, then=alors, else=sinon, elif est une contraction de else if)

4. Les boucles (for, while)

Ce sont des structures itératives qui permettent de répéter un groupe de commandes un certain nombre de fois. Elles peuvent être employées dans des procédures.

4.1 For

Lorsqu'on sait à l'avance le nombre de répétitions, on utilise ***for***, dont la syntaxe est :

for i from début ***to*** fin ***do*** instructions ***end do*** ;

Cela signifie: pour `i` allant de "début" à "fin", exécuter les "instructions" (`i` est une variable choisie par l'utilisateur). En général, "début" et "fin" sont des nombres entiers. Par exemple, voici une boucle ***for*** qui, pour `i` variant de 1 à 10, affiche `i!` .

```
> for i from 1 to 10 do
    i!
end do;
```

1
2
6
24
120
720
5040
40320
362880
3628800 (15)

Par défaut, le pas est 1. On peut spécifier un autre pas d'incrément à l'aide de ***by***. Par exemple, si on décide d'aller de 2 en 2 :

```
> for i from 1 to 10 by 2 do
    i!
end do;
```

1
6
120
5040
362880 (16)

4.2 While

On utilise la boucle ***while*** quand on doit déterminer "en cours de route" le nombre de répétitions. Sa syntaxe est :

while condition *do* instructions *end do* ;

La condition est une expression booléenne et les instructions sont des commandes Maple. Maple évalue l'expression booléenne condition. Tant qu'elle est vraie, il exécute les instructions. Quand elle est fausse, il passe à la suite (après le *end do*).

(rappel pour les non-anglophones : while = tant que)

Par exemple, on souhaite calculer le plus petit entier n tel que la somme des entiers de 1 à n soit supérieure ou égale à 1000.

```
> somme:=0: a:=0:
  while somme<1000 do
    a:=a+1;
    somme:=somme+a;
  end do:
  a;
```

45

(17)

5. La récursivité

Maple accepte des fonctions et des procédures *récursives*, c'est-à-dire qui s'appellent elles-mêmes. Soyez particulièrement soigneux lorsque vous utilisez la récursivité, c'est une source d'erreurs importante. Il faut notamment prévoir un "**cas d'arrêt**" (un cas particulier sans appel récursif) et être sûr qu'il se réalisera ; faute de quoi la récurrence ne s'arrête jamais !

Par exemple, la factorielle est définie sur les entiers naturels par la récurrence : $n! = n * (n-1)!$ avec $0! = 1$. On peut programmer la factorielle dans une procédure récursive :

```
> factorielle:=proc(n)
  if n=0 then RETURN(1)    # cas d'arret
  else RETURN(n*factorielle(n-1)) # on appelle la procédure avec
  l'entrée n-1
  end if;
end proc;
```

```
> factorielle(0); factorielle(10);
```

1

3628800

(18)

```
> factorielle(-1);
```

Error, (in factorielle) too many levels of recursion

Voilà le message d'erreur qu'on obtient quand aucun cas d'arrêt n'a été prévu !

On aurait également pu définir la factorielle dans une fonction :

```
> f:=n->n*f(n-1);    # relation de récurrence
```

$f := n \rightarrow n f(n-1)$

(19)

```
> f(0):=1;    # initialisation
```

$f(0) := 1$

(20)

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

3628800

(21)

Un autre exemple : la suite récurrente définie par :

si $n \leq 0$, $u_n = 3$

si $0 < n$, $u_n = \frac{4u_{n-1} + 1}{5}$

> **u:=proc(n)**

if n<=0 then RETURN(3) else RETURN((4*u(n-1)+1)/5) end if;

end proc;

> **u(-3); u(0); u(5); u(15);**

3

3

5173

3125

32665061773

30517578125

(22)