V. Ripoll

vripoll@dma.ens.fr

MK1 "Calcul formel" Maple

TP3 : Programmation

N.B. : Ce TP est largement inspiré des TPs de Cécile Armana et Marie Albenque.

But du TP3
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.

Et surtout, n'oubliez pas de vous (et de me) poser des questions !

Rappels:

Quelques remarques importantes au vu des premiers TPs:

1/ L'*affectation* d'une variable (c'est-à-dire le fait de donner un nom à une variable) est obtenu par la syntaxe suivante:

```
> a:=2;
```

$$a := 2 \tag{1}$$

Il est important dans les exercices de nommer les variables importantes afin de pouvoir les réutiliser dans la suite.
Attention : il ne faut pas confondre l'opérateur d'affectation := avec le test d'égalité = (que l'on verra dans ce TP). Si vous obtenez un résultat inattendu, c'est souvent à cause d'une faute de frappe : vérifiez que vous avez bien écrit le signe ":=" .

2/ Commencer chaque feuille Maple et chaque exercice par :

```
> restart;
```

Cela permet de vous assurer que toutes les variables sont bien désaffectées et évite beaucoup d'erreurs.

3/ **Différence entre fonctions et expressions:**
Maple autorise la définition de fonctions. La syntaxe est la suivante:

```
> x-> x^2+1:
> f:= x-> cos(x)+sin(x)+1;
```

$$f := x \to \cos(x) + \sin(x) + 1 \tag{2}$$

On accède à la valeur de la fonction en un point comme ceci:

```
> f(Pi);
```

$$0 \tag{3}$$

Attention, fonctions et expressions sont deux choses différentes !

```
> f1:= x^2+x; # f1 est une expression
  f2:= x -> x^2+x; #f2 est une fonction
```

$$f1 := x^2 + x \tag{4}$$
$$f2 := x \to x^2 + x$$

Il ne faut donc pas utiliser la meme syntaxe pour évaluer f1 et f2 en un point :

```
> subs(x=2,f1): # on substitue x par 2 dans l'expression f1
> f2(2):
```

On peut néanmoins passer d'une expression à une fonction, et vice-versa :

```
> f:= t -> t+1: # fonction
> p:= f(x);
```

$$p := x + 1 \tag{5}$$

```
> restart;
> p:=x+1: # expression
> f:= unapply(p,x); # fonction
```

$$f := x \to x + 1 \tag{6}$$

Il n'est pas plus difficile de définir les fonctions de plusieurs variables:

```
> g:= (x,y,z) -> x+3*y+1/z:
> g(2,1,1);
```

$$6 \tag{7}$$

Une fois que vous avez bien lu tous ces rappels et posé toutes vos questions, vous pouvez passer au reste du TP.

1. Les booléens

On appelle *expression booléenne* (de *George Boole*, mathématicien logicien) 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 construire des expressions booléennes.

1.1 Les opérateurs de comparaison

On peut former une expression booléenne en comparant deux expressions de meme 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;
```

$$bool := 1 < 2 \quad (8)$$

```
> evalb(bool);
```

$$true \quad (9)$$

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

$$false \quad (10)$$

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

```
> evalb(x>=y);
```

$$y \le x \quad (11)$$

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) );
```

$$false \quad (12)$$

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);
```

$$false \quad (13)$$

```
> 1<2 and 4<3;
```

$$false \quad (14)$$

1.3 Les fonctions booléennes

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

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

$$true \quad (15)$$

On peut aussi citer 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);
```

$$false \quad (16)$$

(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;
```

$$différence := \mathbf{proc}(x, y)\ RETURN(x - y)\ \mathbf{end\ proc} \quad (17)$$

```
> différence(4,5);
```

$$-1 \quad (18)$$

(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* signifie que la définition de la procédure est terminée.

Voici une procédure que j'appelle *sommeproduit*, qui prend en entrées trois nombres *a*, *b* et *c*, et qui renvoie la liste formée de leur somme et de 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:
> sommeproduit(2,3,4);
```

$$[9, 24] \quad (19)$$

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. Vérifiez votre algorithme à la main, sur des exemples faciles. 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. Souvent Maple affiche un avertissement s'il y a une erreur de syntaxe. Ne pas oublier " end; " à la fin de la procédure.

3. Le test (if)

La structure *if* permet de tester si une condition est vérifiée. Elle peut etre

employée dans une procédure. Elle commence par un *if* et se termine par un *fi*. (*if* à l'envers). 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) fi ;
  end:
> minimum(7,-2);
```

$$-2 \qquad (20)$$

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 *fi* ;

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 *fi*).
* Pour un choix binaire :

if condition *then* instructions_1 *else* instructions_2 *fi* ;

Maple évalue d'abord l'expression booléenne "condition_1". Si le résultat est *true*, alors les instructions_1 sont effectuées. Sinon, Maple effectue les instructions_2.

* 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)
fi ;

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 *fi*). Si c'est *false*, Maple évalue l'expression booléenne condition_2 et procède de meme. 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 etre 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 *od* ;

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. Attention! : de meme qu'un *if* doit terminer par un *fi*, un *do* doit terminer par un *od*. Par exemple, voici une boucle *for* qui, pour i variant de 1 à 10, affiche i!.

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

$$\begin{gathered} 1 \\ 2 \\ 6 \\ 24 \\ 120 \\ 720 \\ 5040 \\ 40320 \\ 362880 \\ 3628800 \end{gathered} \qquad (21)$$

Par défaut, le pas est 1. On peut spécifier un autre pas d'incrémentation 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!
  od;
```

$$\begin{gathered} 1 \\ 6 \\ 120 \\ 5040 \\ 362880 \end{gathered} \qquad (22)$$

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 *od ;*

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 ex"cute les instructions. Quand elle est fausse, il passe à la suite (après le *od*).
(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. Prenez le temps de lire et de bien comprendre cet exemple :

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

45 (23)

5. La récursivité

Maple accepte des fonctions et des procédures *récursives*, c'est-à-dire qui s'appellent elles-memes. Soyez particulièrement soigneux lorsque vous utilisez la récursivité, c'est une source d'erreurs importante. Il faut notamment prévoir un "cas d'arret" (un cas particulier sans appel récursif) et etre sur qu'il se réalisera ; faute de quoi la récurrence ne s'arrete 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
  fi;
  end:
```

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

1 (24)

3628800

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

```
Error, (in factorielle) too many levels of recursion
```

Voilà le message d'erreur qu'on obtient quand aucun cas d'arret n'a été prévu ! (comprendre ce qu'il se passe dans ce cas)

On aurait également pu définir la factorielle dans une fonction :

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

$f := n \to n\, f(n-1)$ (25)

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

$f(0) := 1$ (26)

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

3628800 (27)

Un autre exemple : la suite récurrente définie par :
si $n \le 0$, $u_n = 3$
si $0 < n$, $u_n = \frac{4\,u_{n-1}+1}{5}$

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

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

3 (28)

3

$\frac{5173}{3125}$

$\frac{32665061773}{30517578125}$