Tests Boucles et Procédures

Maple peut aussi être considéré comme un langage de programmation avec certains avantages (facilité de mise en oeuvre) mais loin d'être aussi efficace que le Pascal ou le C.

Les tests

On peut tester plein de choses avec Maple : positif, négatif, plus grand, plus petit et les types des variables avec l'instruction
is

```
> is(ln(3)>1);
```

true

```
> is(ln(3), integer);
```

false

```
> is(x^2<0);
```

FAIL

La réponse FAIL signifie peut-être ben que oui, peut-être ben que non, ou plus précisement : il existe des *x* pour lesquels c'est faux et d'autre pour lequel c'est vrai.
ATTENTION : dans les tests celà implique que *false* n'est pas le contraire de *true*

Les affirmations

On peut préciser à Maple que *x* est réel ou positif, ou entier en utilisant l'instruction assume. Pour rappeler que *x* est soumis
à condition, Maple l'écrira toujours suivi d'un tilde (on peut l'éviter si on le souhaite)

```
> assume(x ,real):is(x^2<0);
```

false

```
> y := sqrt(1+x^2);
```

$$y := \sqrt{1 + x\sim^2}$$

```
> is(y > 0);
```

true

On peut imposer plusieurs conditions, la première définie par assume, les autres avec additionally :

```
> assume(k > 0) : additionally(k < 1): is(k*(1-k) > 0);
```

true

On peut connaitre le statut d'une variable *x* en utilisant l'instruction about

```
> about(k);about(y);
Originally k, renamed k~:
  is assumed to be: RealRange(Open(0),Open(1))

y:
  nothing known about this object
```

Les structures de contrôle.

Voyons maintenant les boucles

```
> for i from 1 to 5 do z := sqrt(i); od:
> z;
```

$$\sqrt{5}$$

Si votre boucle instruction se termine par : elle ne provoquera aucun affichage bien qu'elle soit exécutée. Pour afficher les résultats intermédiaires terminez par ;
On peut executer plusieurs instructions dans une même boucle.

```
> for i from 1 to 5 do z := sqrt(i): od;
```

$$z := 1$$

$$z := \sqrt{2}$$
$$z := \sqrt{3}$$
$$z := 2$$
$$z := \sqrt{5}$$

On peut aussi forcer l'affichage avec une instruction print , ou printf exactement comme en C.

```
> for i from 1 to 5 do z := sqrt(i): print(z,evalf(z)) od:
```

$$1, 1.$$
$$\sqrt{2}, 1.414213562$$
$$\sqrt{3}, 1.732050808$$
$$2, 2.$$
$$\sqrt{5}, 2.236067978$$

Par défaut l'incrément est de 1, mais il peut être spécifié ; sans autre précision on part de 1 et on va de 1 en 1.

```
> for i from 1/2  to 12 by 3/2 do i; od:
```

Les valeurs désignées par from, to, by ne sont pas nécéssairement entières mais doivent être numériques...sinon
Maple proteste ! Contrairement à ce que vous pensez Pi et sqrt(2) par exemple ne sont pas numériques !!!

```
> for i from 0 to 6 by Pi/6 do sin(i); od;
Error, unable to execute for statement
```

```
> for i from 0 to 6 by evalf(Pi/6) do sin(i); od;
```

$$0$$
$$.5000000002$$
$$.8660254042$$
$$1.$$
$$.8660254030$$
$$.4999999983$$
$$-.2410206762\ 10^{-8}$$
$$-.5000000024$$
$$-.8660254054$$
$$-1.$$
$$-.8660254018$$
$$-.4999999962$$

Conclusion : evalf() retourne bien une valeur de type numérique, par contre les calculs trigo sont faits en virgule flottante et non avec les valeurs exactes ; il serait bon de récrire la boucle pour n'utiliser que des entiers

```
> for i from 0 to 6 do cos(Pi*i/6); od;
```

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

$$\frac{-1}{2}$$

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

-1

A titre de deuxième exemple cherchons le plus petit entier tel que (11/10)^n > 3. (on oublie la fonction log !)
On fait afficher à Maple toutes les puissances de 11/10 et on choisit nous-mêmes : vraiment inélégant.

```
> x := 11/10 : for i to 30 do i;x^i; od:
```

Il y a deux façons de demander à Maple de s'arrêter de lui-même :
Avec une instruction while + *condition* : on execute la boucle tant que *condition* est satisfaite ou que la borne to est atteinte

```
> for i to 30 while x^i < 3 do i; od;
```

1
2
3
4
5
6
7
8
9
10
11

```
> i;x^i;
```

12

$$\frac{3138428376721}{1000000000000}$$

Remarquez qu'à la fin de la boucle *i* est incrément par rapport à la borne ou la derniere valeur utilisée.
Deuxième méthode :on sort de la boucle en utilisant l'instruction break généralement associée à un test

if..then...else

La forme générale est : if condition then instructions else instructions fi
Dans le cas de tests imbriqués on peut aussi utiliser le mot-clé elif (contraction de else if).

```
> for i do
  if (x^i > 3) then break fi;
  od;
```

Remarque : pour écrire sur plusieurs lignes entre do et od utiliser MAJUSCULE + Entrée au lieu de Entrée en fin de ligne.

```
> i,x^i;
```

$$12, \frac{3138428376721}{1000000000000}$$

Dans le cas d'une sortie prématurée, la valeur de *i* en sortie est la bonne, i.e. non incrémentée.

Ici la boucle utilisait la valeur affectée à *x* . On peut affecter une autre valeur à *x* et réécrire la boucle ou bien trouver moyen de passer *x* en paramètre dans l'instruction.

Les procédures

En Maple cela s'appelle une procédure, généralisation de la notion de fonction.

```
> mafonction := proc(x) sqrt(1+x^2) end;
```

mafonction := **proc**(*x*) sqrt(1 + *x*^2) **end proc**

On a défini une procedure appelée mafonction ayant un seul argument *x* et qui retourne la valeur sqrt(1+x) : on a fait exactement l'équivalent de mafonction := x -> sqrt(1+x^2).

```
> mafonction(sqrt(3));
```

2

En Maple une procédure retourne toujours une valeur, généralement la dernière calculée ou affichée, mais on peut forcer cette valeur avec l'instruction RETURN.

```
> monmax := proc(x,y)
  if (x>y) then RETURN(x) fi;
  if (y>x) then RETURN(y) fi;
  'egalité';
  end;
```

monmax := **proc**(*x*, *y*)

if *y* < *x* **then** RETURN(*x*) **end if**; **if** *x* < *y* **then** RETURN(*y*) **end if**; *'egalité'*

end proc

```
> monmax(1,2);
```

2

```
> monmax(14,12);
```

14

```
> monmax(2,sqrt(4));
```

egalité

En Maple une procédure peut retourner à peu prés n'importe quoi...des chaînes de caractère, des séquences, des matrices...

En Maple les chaînes de caractères sont entre ' ' (pour les Release < 5) et entre " " pour les plus récentes. Attention : il s'agit de l'accent inversé ALTGR+7 (suivi d'un espace éventuellement), et non de l'apostrophe !!

Si on essaye la procedure avec un *x* complexe, on obtient ceci :

```
> monmax(3+4*I,2);
Error, (in monmax) cannot evaluate boolean
```

En fait le premier test retourne FAIL et Maple ne peut évaluer le résultat qui doit être *true* ou *false*.

Ecrivons une procédure, de paramètre *i* et *a*, calculant le plus petit *n* tel que (1+i)^n > a. (i est censé représenter un taux d'interêt)

```
> myproc := proc(i,a) local x,n;
  x := 1+i;
  for n do if (x^n > a) then break; fi;od;
  RETURN(n);
  end;
```

myproc := **proc**(*i*, *a*)

local *x*, *n*;

x := 1 + *i*; **for** *n* **do if** *a* < *x*^*n* **then break end if end do**; RETURN(*n*)

end proc

```
> myproc(1/10,3);
```

$$12$$

```
> myproc(1,25);
```

$$5$$

Le mot clé local signifie que, dans la procedure *x* et *n* sont des variables locales : connues uniquement dans celle-ci et sans rapport avec les variables que connaissait Maple au moment d'executer la procédure et qui sont dites *globales* (mot-clé global).

```
> x;n;
```

$$\frac{11}{10}$$

$$n$$

Au cas bien probable où cela ne marche pas du premier coup, il existe toute une batterie de moyens de contrôle.
debug() permet l'execution pas à pas et détaillée de la ou des procédures passées en argument ; utiliser undebug pour revenir à la normale. Un synonyme de debug est trace, mais ce nom est redéfini avec le package d'algèbre linéaire.

```
> debug(myproc);
```

$$myproc$$

```
> myproc(1/10,5);
{--> enter myproc, args = 1/10, 5
```

$$x := \frac{11}{10}$$

```
<-- exit myproc (now at top level) = 17}
```

$$17$$

Guère convaincant ici, car on n'a pas le détail de la boucle for : on pourrait rajouter l'affichage de *n* dans la boucle for.

```
> undebug(myproc):
```

Une autre solution consiste à utiliser la variable printlevel qui fixe le niveau d'affichage de Maple

```
> printlevel := 1000:
> myproc(1/10,5);
{--> enter myproc, args = 1/10, 5
```

$$x := \frac{11}{10}$$

```
<-- exit myproc (now at top level) = 17}
```

$$17$$

```
> printlevel := 1:  #retour à la normale
```

L'instruction next : dans une boucle comportant plusieurs instructions on a souvent besoin de ne pas exécuter certaines instructions et passer à la valeur suivante (à la différence de break qui termine définitivement).
Exemple : on donne un entier n assez grand, et on compte la fréquence des valeurs de cos(k), k entier naturel, k < n, qui sont positives.

```
> ergo := proc(n) local k,compteur;
  compteur := 0;
  for k from 0 to n do
  if is(cos(k) < 0) then next; fi;  # afin d'éviter de compter les
  négatifs.
  compteur := compteur+1;
  od;
  print('il y a',compteur, 'cosinus positifs');
  evalf(compteur/k);
```

```
   end;
>
```

ergo := **proc**(*n*)

local *k, compteur*;

compteur := 0;

for *k* **from** 0 **to** *n* **do if** is(cos(*k*) < 0) **then next end if**; *compteur* := *compteur* + 1 **end do**;

print(‘*il y a*‘, *compteur*, ‘*cosinus positifs*‘);

evalf(*compteur* / *k*)

end proc

C’est un exemple : il aurait été possible (et plus logique !) d’incrémenter le compteur dans le test plutôt qu’à l’extérieur. Le test avec (cos(k) < 0) conduit à une erreur : cos(k) n’est pas numérique, d’où l’utilisation de is. Remarque : comme Pi n’est pas rationnel, on teste l’inégalité stricte !

```
> ergo(25);
```

il y a, 14, *cosinus positifs*

.5384615385

Une dernière astuce : on peut demander à Maple d’évaluer le temps de calcul avec la fonction time() qui retourne le temps (en secondes) écoulé depuis le debut de la session

```
> debut := time() :ergo(1000); time()-debut;
```

il y a, 502, *cosinus positifs*

.5014985015

24.604

Si vous faites tourner ce programme pour la première fois : il a mis environ 24 secondes, alors que c’est quasi-instantané la deuxième !

```
> debut := time() :ergo(1000); time()-debut;
```

il y a, 502, *cosinus positifs*

.5014985015

.730

Explication : toute fonction Maple, comme la fonction cosinus, dispose d’une table de remember, où elle stocke les valeurs
calculées (on verra plus tard comment profiter de ceci) : la deuxième fois Maple n’a eu qu’à faire les tests et non pas les
calculs des cosinus. Vous pourrez constater en regardant en bas à droite que l’espace mémoire utilisé a nettement augmenté !
Question subsidiaire : pourquoi avoir appelé cette procédure ergo ?

Exercice