FASCICULE DE PROGRAMMATION

LAURENT POINSOT

1. Introduction

Le but de ce fascicule est de fournir aux étudiants les bases de la programmation sous Maple afin de réaliser les exercices qui leurs sont proposés. Le cours « magistral » sur la programmation viendra plus tard...

Sous Maple, comme dans la grande majorité des langages de programmation, il existe trois instructions fondamentales pour la programmation, à savoir, `if`, `for` et `while`. Par ailleurs il existe bien évidemment la possibilité d'écrire des procédures qui permettent de « factoriser » des parties de programmes.

La combinaison des touches [Maj.] (parfois désignée [Shift] ou représentée par une flêche « montante ») et [Entrée] permet de passer à la ligne sans exécuter la commande. On peut donc écrire plusieurs lignes dans le même « groupe d'instructions », lesquelles sont ensuite exécutées conjointement avec la touche [Entrée]. Nous utiliserons cela pour écrire des programmes avec procédures et / ou instructions d'itération (ou conditionnelles).

Dans la syntaxe des instructions de programmation <objet> signifie que l'objet entre $<$ et $>$ est un nom général que l'on devra remplacer afin d'écrire l'instruction, et [instruction] signifie que l'instruction entre crochets est facultative. Par ex. <var> désigne un nom de variable quelconque que l'utilisateur prendra soin de remplacer par un vrai nom de variable valide sous Maple.

2. Procédures

La syntaxe d'une procédure Maple est la suivante :

```
<Nom de la procédure> := proc ([<argument1>,<argument2>,...])
[local <var1>,<var2>,... ;]
<instruction1> ;
```

```
<instruction2> ;
...

[return <résultat> ;]
end proc ;
```

(1) `<Nom de la procédure>` est le nom que l'on donne à la procédure ;
(2) La procédure utilisera les valeurs données par l'utilisateur pour les arguments **formels** `<argument1>,<argument2>,...` s'ils sont présents (une procédure sans argument formel se déclare donc par `<Nom de la procédure> := proc ()`... ne pas oublier les parenthèses !) ;
(3) Les variables **locales** `<var1>,<var2>,...`, elles aussi facultatives, doivent être déclarées après le mot clef `local`. Elles ne sont accessibles et manipulables que dans le bloc d'instructions de la procédure à partir de l'endroit où elles sont déclarées et jusqu'à l'instruction finale « `end proc ;` » ;
(4) L'instruction « `return` » permet de renvoyer le résultat d'un calcul effectué dans la procédure et qui est contenu dans la variable `<résultat>` ;
(5) La procédure se termine par l'instruction « `end proc ;` ».

Un premier exemple : la procédure suivante effectue l'addition des nombres a et b passés en argument :

```
addition := proc(a,b)
local res ;
res :=a+b ;
return res ;
end proc ;
```

Remarquons que l'on pourrait aussi écrire cette même procédure de la façon suivante :

```
addition := proc(a,b)
return (a+b) ;
end proc ;
```

Voici comment on utilise la procédure :

> addition (10,7) ;

17

> toto := addition (2,3) ;

toto := 5

> res ;

res

Lors de l'appel de la procédure « addition » par l'instruction « addition (10,7) ; », l'argument formel a prend la valeur 10 et l'argument formel b prend la valeur 7, et tous les calculs effectués dans « addition » sont faits avec a valant 10 et b valant 7. On récupère la valeur calculée par la procédure « addition » dans la variable « toto » grâce à l'instruction « return » (sans cette dernière, on ne peut affecter une variable avec le résultat d'une procédure !). Par ailleurs on remarque que la variable « res » est inaccessible à l'extérieur de la procédure « addition » car c'est une variable locale à cette dernière.

Quelques remarques (très) importantes :

(1) Un argument formel ne peut pas être modifié à l'intérieur d'une procédure. Par ex. :

```
> toto := proc(n)
n :=n+1 ;
return n ;
end proc ;

> toto (3) ;
```

Error, in (toto) illegal use of a formal parameter.

Si on veut faire cela, il faut utiliser une variable locale :

```
> toto := proc(n)
local m ;
m :=n+1 ;
return m ;
end proc ;

> toto (3) ;
```

(2) On peut définir une procédure sans argument formel : par exemple une procédure qui ne fait que renvoyer « 0 » :

```
> zero := proc()
return 0;
end proc;

> zero ();
```

0

(3) Dès qu'une instruction « return » est exécutée dans une procédure, on sort de celle-ci, et l'exécution du programme reprend après le « end proc ; » final. Par ex. :

```
> toto := proc(a,b,c)
return -a;
return ((a+b)*c);
end proc;

> toto (1,3,7);
```

-1

Dans cet exemple on voit que seul le premier « return » est effectué et pas le second.

Un autre exemple de procédure : la procédure suivante crée la liste des produits $i*x$ où i varie de 0 jusqu'à l'entier n et x est une variable :

```
> liste := proc (n)
local L,i;
L :=[seq(i*x,i=0..n)];
return L;
end proc;

> liste (5);
```

[0,x,2x,3x,4x,5x]

3. Instruction conditionnelle `if`

La syntaxe de l'instruction conditionnelle `if` est la suivante :

```
if <condition> then
<instruction1> ;
<instruction2> ;
```

```
...
[else
<instructionA> ;
<instructionB> ;
...]
end if ;
```

(1) Si la <condition> est vraie (c'est-à-dire qu'elle est égale au booléen `true`), les instructions <instruction1>, <instruction2>, ..., sont exécutées. Puis l'exécution reprend après l'instruction finale « end if ; » ;

(2) Si la <condition> est fausse (autrement dit elle a pour valeur `false`), l'exécution se poursuit directement après l'instruction finale «end if ; » ;

(3) Toutefois, si l'instruction optionnelle « else » est spécifiée, et toujours dans le cas où <condition> est fausse, les instructions <instructionA>, <instructionB>, ..., sont exécutées. Puis l'exécution du programme reprend après l'instruction finale « end if ; » ;

(4) L'instruction conditionnelle `if` se termine par l'instruction « end if ; ».

Voici un exemple qui teste si un entier est pair ou non :

```
> pair := proc (n)
local result ;
if n mod 2 = 0 then
result := true ;
else
result := false ;
end if ;
return result ;
end proc ;

> pair (6) ;
                                    true
> pair (15) ;
                                   false
```

4. Boucle `for`

La syntaxe de l'instruction d'itération `for` est la suivante :

```
for <compteur> from <début> to <fin> [by <pas>] do
<instruction1> ;
<instruction2> ;
...
end do ;
```

(1) La variable entière <compteur> est incrémentée (de +1), en partant de l'entier <début> et jusqu'à l'entier <fin> inclus ;

(2) Si l'instruction optionnelle `by` est spécifiée, la variable <compteur> est incrémentée de la valeur <pas> ;

(3) Pour chaque valeur de la variable <compteur>, les instructions <instruction1>, <instruction2>, ... sont exécutées ;

(4) Une fois que <compteur> est strictement supérieur à <fin>, la boucle s'arrête et l'exécution du programme reprend après ce qui suit l'instruction finale « end do ; ».

Un petit exemple qui calcule à l'aide d'une boucle `for` la somme $\sum_{i=1}^{k} i * n = 1 * n + 2 * n + \cdots + k * n$ où les entiers n et k sont choisis par l'utilisateur :

```
> somme := proc (n,k)
local S,i ;
S :=0 ;
for i from 1 to k do
S :=S+i*n ;
end do ;
return S ;
end proc ;

> somme (2,3) ;
                    12
```

En détail : quand on « entre » dans la boucle, S vaut 0 et i prend la valeur 1. Puis on calcule $S+i*n = 0+1*2 = 2$ que l'on affecte à S (qui vaut donc maintenant 2). Puis on passe au pas suivant : $i := i + 1 = 2$ et on calcule $S := S + i * n = 2 + 2 * 2 = 6$. Puis on passe au pas suivant : $i = 3$ et on calcule $S := S + i * n = 6 + 3 * 2 = 12$. Puisqu'on est arrivé à $i = 3 = k$, on arrête l'exécution de la boucle `for` et on reprend l'exécution du programme après le « end do ; » soit dans notre exemple, on exécute « return S ; ».

5. Boucle d'itération `while`

La syntaxe de la boucle `while` est la suivante :

```
while <condition> do
<instruction1> ;
<instruction2> ;
...
end do ;
```

(1) Les instructions <instruction1>, <instruction2>, ..., sont exécutées tant que la <condition> est vraie ;

(2) Dès que la <condition> est fausse, l'exécution du programme se poursuit après l'instruction finale « end do ; ».

Voici un petit exemple qui calcule la somme des entiers strictement inférieurs à un entier n : $\sum_{i=1}^{n-1} i$:

```
> somme := proc (n)
local S, i ;
i := 1 ;
S := 0 ;
while i<n do
S := S+i ;
i := i+1 ;
end do ;
return S ;
end proc ;
```

Cette même procédure écrite avec une boucle `for` en lieu et place de la boucle `while` prend la forme suivante :

```
> somme := proc (n)
local S,i ;
S := 0 ;
for i from 1 to n-1 do
S := S+i ;
end do ;
return S ;
end proc ;
```

On remarque donc que dans un `while` il est nécessaire d'initialiser le compteur de la boucle (dans l'exemple : $i := 1$) et de gérer soi-même l'incrémentation de ce compteur (dans l'exemple : $i := i + 1$) ce qui est fait automatiquement dans une boucle `for`.

Notons que si la <condition> de la boucle `while` est fausse **avant** d'entrer dans la boucle `while`, celle-ci n'est jamais exécutée.

6. Opérateurs booléens (ou logiques)

Une expression booléenne (ou logique) est une expression qui a pour valeur un booléen, soit `true`, soit `false`. Par exemple la <condition> dans les instructions `if` et `while` est une expression booléenne. On peut obtenir de nouvelles expressions booléennes à partir d'expressions booléennes données en les « connectant » à l'aide d'opérateurs logiques que sont le « et » (`and`), le « ou » (`or`) et le « non » (`not`). C'est très utile justement pour écrire des conditions complexes dans les instructions `if` ou `while`.

La commande `evalb` (signifiant en anglais « EVALuate to Boolean ») détermine si une expression booléenne est vraie ou fausse :

```
> evalb ((1 < 4) and (3 <> 3)) ;
                                false
> evalb ((4 < 1) or (5 = 5)) ;
                                true
> evalb (not (2 <= 3)) ;
                                false
```