CHAPITRE 3:
LA PROGRAMMATION

Remarque importante: pour pouvoir entrer plusieurs lignes d'instructions Maple successives, sans exécution à la fin de chaque ligne, presser ***Shift+Entrée*** à la fin de chaque ligne.
Puis, pour valider l'ensemble des lignes ainsi écrites, presser ***Entrée*** en plaçant le curseur par exemple sur la dernière ligne écrite.

LES STRUCTURES DE CONTRÔLE

Structure de contrôle if .. then:

if condition1 ***then*** instruction 1
elif condition2 ***then*** instruction2 (***elif*** et ***else*** sont optionnels)
elif condition3 ***then instruction3 ...***
else instructionN
end if;

Effectue un test selon certaines conditions: condition1, condition2, ..., conditionN.
elif signifie "autrement si" . La dernière instruction à exécuter doit être précédée de ***else***.

Exemple 1: résolution de l'équation du premier degré $ax + b = 0$.
Modifiez les valeurs de a et de b, puis validez les lignes suivantes:

```
> a:=7:b:=3:
> if a<>0 then print(`Une solution : x `=-b/a)
    elif b=0 then print(`Tout x est solution`)
    else print(`Pas de solution`)
  end if;
```

$$\textit{Une solution : } x = \frac{-3}{7}$$

Structure de contrôle while .. do:

while condition ***do*** ... ***end do;***
Exécute une suite d'instructions, tant que la condition est vraie.

Exemple 2: calcul de la somme des 100 premiers entiers naturels

```
> somme:=0:k:=-1:
  while k<100 do k:=k+1:somme:=somme+k end do:
  `somme `=somme;
```

$$somme = 5050$$

Structure de contrôle for .. to:

for variable ***from*** initiale ***to*** finale ***by*** pas ***do*** ... ***end do;***

Exécute une boucle pour une variable allant d'une valeur initiale à une valeur finale, avec un pas donné.

from , by peuvent être omises ou écrites dans un ordre quelconque.
Si l'on omet ***from*** , initiale vaut 1. Si l'on omet ***by*** , pas vaut 1.

Exemple 3: calcul de la somme des 100 premiers entiers naturels

```
> somme:=0:
  for k from 0 to 100 do somme:=somme+k end do:
  `somme `=somme;
```

somme = 5050

Exemple 4: calcul de la somme des entiers impairs inférieurs à 100
Ici, on omet ***from*** donc la première valeur de *k* est 1. La dernière valeur prise par *k* sera 99.

```
> somme:=0:
  for k to 100 by 2 do somme:=somme+k end do:
  `somme `=somme;
```

somme = 2500

for variable ***in*** expression ***do*** .. ***end do;***

Exemple 5: donner les nombres premiers de la liste [31,39,47,105]

```
> for k in [31,39,47,105] do
    if isprime(k) then print(k,`est premier`) end if
  end do;
```

31, *est premier*

47, *est premier*

LES PROCÉDURES:

Un programme MAPLE peut être organisé en sous-programmes appelées *procédures*.
Une procédure, de type *procedure*, est définie par le mot-clé ***proc*** et peut être assignée à un nom de variable. Pour définir une procédure intitulée *nom* , on utilisera la syntaxe suivante:

nom:=***proc***(paramètres_formels)
  global variables_globales; (la ligne ***global*** est optionnelle)
  local variables_locales; (la ligne ***local*** est optionnelle)
  description chaîne_de_description; (la ligne ***description*** est optionnelle)
  option nom_option; (la ligne ***option*** est optionnelle)
  . . . instructions . . . (corps de la procédure)
end proc;

Exemple 6:
La procédure intitulée *maxi* calcule le plus grand de 2 entiers positifs.
Elle comporte 2 paramètres formels *u* et *v*. Leur type peut être omis, mais si on le veut le préciser,
bien noter la syntaxe ***u* :: *posint*** , ***v* :: *posint*** pour signifier que *u* et *v* sont de type *posint*.
Elle ne comporte ni variables globales, ni variables locales, mais une chaîne de description de la procédure et une option de copyright.

```
> maxi:=proc(u::posint,v::posint)
  description "calcule le plus grand de 2 entiers positifs";
  option `Copyright (c) 2001  A.Le Stang`;
    if u<v then v else u end if;
  end proc;
```

maxi :=

proc(*u*::*posint*, *v*::*posint*) **description** "calcule le plus grand de 2 entiers positifs" ... **end proc**

```
> maxi(5,9);maxi(1.2,Pi);
```

9

```
Error, invalid input: maxi expects its 1st argument, u, to be of type posint,
but received 1.2
```

Une ***variable locale*** est une variable qui n'est reconnue qu'à l'intérieur de la procédure, contrairement à une ***variable globale*** accessible en dehors de la procédure.
Exemple 7:

```
> messages:=proc()
  global message1;
  local message2;
    message1:="Hello, world!";
    message2:="How are you?";
  end proc;
```

messages := **proc**()

local *message2*;

global *message1*;

message1 := "Hello, world!"; *message2* := "How are you?"

end proc

```
> messages();
```

"How are you?"

```
> message1,message2;
```

"Hello, world!", *message2*

La valeur de la variable locale *message2* n'a pas été reconnue, contrairement à celle de la variable globale *message1*.

Remarque importante:
Un paramètre formel passé à une procédure ne peut être modifié à l'intérieur de cette procédure

Exemple 8: Diviser un entier positif *x* par 2 tant qu'il est pair.

```
> div:=proc(x::posint)
    while type(x,even) do x:=x/2 end do;
  end proc;
```

div := **proc**(*x*::*posint*) **while** type(*x*, *even*) **do** *x* := 1 / 2∗*x* **end do end proc**

```
> div(48);
Error, (in div) illegal use of a formal parameter
```

La tentative d'affecter au paramètre formel *x* sa valeur divisée par 2 provoque une erreur.
On peut modifier la procédure de la manière suivante:

```
> div:=proc(x::posint)
   local y;
   y:=x;
   while type(y,even) do y:=y/2 end do;
   y
  end proc;
```

div := **proc**(*x*::*posint*) **local** *y*; *y* := *x*; **while** type(*y*, *even*) **do** *y* := 1 / 2∗*y* **end do**; *y* **end proc**

```
> div(48);
```

3

On obtient en effet successivement : 48,24,12,6,3.

```
> div(45);
```

45

return met fin à la procédure en donnant un résultat.
next permet de passer à la valeur suivante dans une itération.
break permet de sortir de la structure de contrôle en cours.

Exemple 9: On parcourt une liste de nombres. Au premier entier positif trouvé, on sort de la procédure
avec pour résultat cet entier, sinon si ce nombre est négatif, on sort de la boucle avec pour résultat -1,
sinon on passe au nombre suivant.

```
> liste:=proc(L::list(numeric))
  local k;
  for k in L do
    print(`lecture de`,k);
    if type(k,nonnegint) then return k
    elif k<0 then break;
    else next;
    end if
  end do;
  return -1;
  end proc;
```

liste := **proc**(*L*::list(*numeric*))
local *k*;
for *k* **in** *L* **do**

```
        print( `lecture de`, k );
        if type( k, nonnegint ) then return k elif k < 0 then break else next end if
    end do;
    return -1
end proc
```

```
> liste([12.56,8.9,7,3.14]);
```

lecture de, 12.56

lecture de, 8.9

lecture de, 7

7

```
> liste([12.56,-8.9,7,3.14]);
```

lecture de, 12.56

lecture de, -8.9

-1

```
> liste([12.56,8.9,7.1,3.14]);
```

lecture de, 12.56

lecture de, 8.9

lecture de, 7.1

lecture de, 3.14

-1

Récursivité:

Une procédure qui fait référence à elle même dans sa définition est une procédure ***récursive*** .

Exemple 10: factorielle de *n*

```
> factorielle := proc(n::nonnegint)
    if n>0 then n*factorielle(n-1) else 1 end if;
  end proc;
```

factorielle := **proc**(*n*::*nonnegint*) **if** 0 < *n* **then** *n*∗factorielle(*n* – 1) **else** 1 **end if end proc**

```
> factorielle(12);
```

479001600

Fonction *error*:

La fonction ***error*** permet de définir des messages d'erreur personnalisés.

Exemple 11: racine carrée de *x*

```
> racine := proc (x)
   if x<0 then error "invalide x: %1", x else simplify(x^(1/2))
  end if
  end proc;
```

racine :=

proc(*x*) **if** *x* < 0 **then error** "invalide x: %1", *x* **else** simplify(sqrt(*x*)) **end if end proc**

```
> racine(144);
```

12

```
> racine(-144);
Error, (in racine) invalide x: -144
```

LES MODULES:

Un ***module*** est une collection de procédures et de données.
Un module, de type *module*, est défini par le mot-clé ***module*** et peut être assigné à un nom de variable. Pour définir un module intitulé *nom* , on utilisera la syntaxe suivante:

nom:=***module***()
  global variables_globales; (la ligne ***global*** est optionnelle)
  local variables_locales; (la ligne ***local*** est optionnelle)
  export variables_locales_exportées; (la ligne ***export*** est optionnelle)
  description chaîne_de_description; (la ligne ***description*** est optionnelle)
  option nom_option; (la ligne ***option*** est optionnelle)
  . . . instructions . . . (corps du module)
end module;

Contrairement au cas des procédures, le module ne peut avoir de paramètres formels .
On retrouve les notions de variables globales et locales, comme pour les procédures, mais les modules peuvent exporter des variables locales (celles de la ligne ***export***), auxquelles on a accès de l'extérieur du module en utilisant la syntaxe suivante:
nom **:-** *var* permet d'accéder à la variable locale exportée *var* du module *nom*.

En utilisant la terminologie de la programmation orientée objet, les variables déclarées dans la ligne ***local*** s'apparentent à des membres *privés* du module, tandis que celles déclarées dans la ligne ***export*** s'apparentent à des membres *publics* du module.

Exemple 12: Anneau Z/4Z des entiers modulo 4.

```
> Z4:=module()
      description "Arithmétique modulo 4";
      export add,mult,opp;
      add := (x,y)-> (x+y) mod 4;
      mult := (x,y)-> (x*y) mod 4;
      opp := x-> (-x) mod 4;
  end module;
```

Z4 := **module**() **export** *add*, *mult*, *opp*; **description** "Arithmétique modulo 4"; **end module**

Le module comporte 3 variables locales exportées *add*, *mult*, *opp*.
Vérifions que: (2+3) mod 4 = 1 , que (2*3) mod 4 = 2 et *opp*(1) = -1 mod 4 = 3 :

```
> Z4:-add(2,3);
```

1

```
> Z4:-mult(2,3);
```

2

```
> Z4:-opp(1);
```

Exemple 13: Anneau Z/*n*Z des entiers modulo *n*.
On désire modifier le programme précédent pour effectuer de l'arithmétique modulo *n*, avec 0 < *n* entier quelconque.
Un module n'acceptant pas de paramètre formel, écrivons une procédure *Construire_Zn* de paramètre formel *n* , rendant pour résultat un module:

```
> Construire_Zn:=proc(n::posint)
    module()
      description "Arithmétique modulo n";
      export add,mult,opp;
      add := (x,y)-> (x+y) mod n;
      mult := (x,y)-> (x*y) mod n;
      opp := x-> (-x) mod n;
    end module;
  end proc;
```

Construire_Zn := **proc**(*n*::*posint*)
module()
export *add*, *mult*, *opp*;
description "Arithmétique modulo n";
add := (*x*, *y*) → (*x* + *y*) mod *n*; *mult* := (*x*, *y*) → *x***y* mod *n*; *opp* := *x* → (–*x*) mod *n*
end module
end proc

```
> Z7:=Construire_Zn(7);
```

Z7 := **module**() **export** *add*, *mult*, *opp*; **description** "Arithmétique modulo n"; **end module**

```
> Z7:-add(6,5);
```

4

```
> Z7:-mult(5,3);
```

1

```
> Z7:-opp(3);
```

4

On peut utiliser la structure de contrôle ***use*** *nom* ***in ... end use***
pour accéder aux variables locales exportées du module *nom* sans avoir recours à l'utilisation de **:-**
Exemple: (opp(3) mod 7) = (4 mod 7) = 4 ((opp(3) mod 7)*5) mod 7 = (4*5) mod 7 = 6
(2 + ((opp(3) mod 7)*5) mod 7) = (2+6) mod 7 = 1

```
> use Z7 in
    add( 2, mult( opp(3), 5) )
  end use;
```

1

La fonction ***member*** permet de savoir si une expression est membre d'un module:

```
> member(mult,Z7) , member(prod,Z7);
```

true, *false*

Le constructeur Record:

Le constructeur ***Record*** crée un *enregistrement* Maple. Un enregistrement, de type *record*, est défini par les noms de ses champs (appelés "slots" ou "fields" en anglais).

Exemple 14: On veut créer un enregistrement pour modéliser les nombres complexes.
Un nombre complexe *z* est défini par 2 champs: sa partie réelle et sa partie imaginaire.

```
> z := Record( 're', 'im' );
```

z := **module**() **export** *re*, *im*; **option** *record*; **end module**

On voit qu'un enregistrement est un module particulier, qui a l'option *record*.

On définit un nouveau type *complexe* associé et la fonction module d'un nombre complexe:

```
> `type/complexe` := 'record( re, im )':
   Mon_module:=(z::complexe)->sqrt(z:-re^2+z:-im^2);
```

$$Mon\_module := z::complexe \rightarrow \sqrt{z:\text{-}re^2 + z:\text{-}im^2}$$

```
> z:-re:=1 : z:-im:=2 :Mon_module(z);
```

$$\sqrt{5}$$

On peut aussi initialiser les valeurs des parties réelle et imaginaire de la façon suivante:

```
> z:= Record( 're' = 1, 'im' = 2 ):Mon_module(z);
```

$$\sqrt{5}$$

LES LIBRAIRIES ET PACKAGES:

Pour obtenir le listing de certaines fonctions des librairies MAPLE, comme ici la fonction *ithprime* utiliser:

```
> interface(verboseproc=2);print(ithprime);
```

proc(*i*) ... **end proc**

Les ***packages*** regroupent des fonctions utiles dans des domaines particuliers.
La fonction suivante appelle la page d'aide donnant les packages existants :

```
> ?index,package
```

La fonction ***with*** permet de charger toute ou partie d'un package choisi:

```
> with(student);
  #charge toutes les fonctions du package student
```

[D, *Diff, Doubleint, Int, Limit, Lineint, Product, Sum, Tripleint, changevar, completesquare, distance, equate, integrand, intercept, intparts, leftbox, leftsum, makeproc, middlebox, middlesum, midpoint, powsubs, rightbox, rightsum, showtangent, simpson, slope, summand,*

trapezoid]

Chargement de la fonction *binomial* du package *combinat :*

```
> with(combinat,binomial);
```

[binomial]

Cette fonction retourne le nombre binomial(n, p) = $\frac{n!}{p!\,(n-p)!}$ appelé aussi C(n, p).

Exercice corrigé 3:

Ex 3.1

Ecrire une procédure calculant le plus grand de 3 entiers naturels non nuls *a,b,c*.

```
> max3:=proc(a::posint,b::posint,c::posint)
    local max2;
      max2:=proc(x::posint,y::posint)
        if x>y then x else y end if;
      end proc;
      max2(a,max2(b,c));
    end proc;
```

max3 := **proc**(*a*::*posint*, *b*::*posint*, *c*::*posint*)

local *max2*;

max2 := **proc**(*x*::*posint*, *y*::*posint*) **if** *y* < *x* **then** *x* **else** *y* **end if end proc**;

max2(*a*, *max2*(*b*, *c*))

end proc

```
> max3(5!,5^3,123);
```

125

max3 comporte une variable locale *max2* à laquelle est affectée une procédure calculant le plus grand de 2 entiers naturels non nuls.

Ex 3.2

1° Écrire un module *pnt* permettant de modéliser un point du plan, défini par ses coordonnées *x*,*y*. Ce module comportera des variables locales exportées rendant comme résultat l'abscisse et l'ordonnée du point.

2° Écrire un module *cercle* permettant de modéliser un cercle du plan, défini par son centre et son rayon. Ce module comportera des variables locales exportées rendant comme résultat le centre, le rayon, le diamètre, l'aire, et la circonférence du cercle.

1° Définition de la procédure *pnt* rendant pour résultat un module:

```
> pnt:=proc(x,y)
    module()
      export abscisse,ordonnee;
      abscisse:=()->x;
```

```
      ordonnee:=()->y;
    end module
  end proc;
```

pnt := **proc**(*x*, *y*)
  module()
  export *abscisse*, *ordonnee*;
    abscisse := ( ) → *x*; *ordonnee* := ( ) → *y*
  end module
end proc

Définition d'un point *A* de coordonnées (*a*,*b*) :

```
> A:=pnt(a,b);
```

A := **module**() **export** *abscisse*, *ordonnee*; **end module**

```
> A:-abscisse(), A:-ordonnee();
```

a, *b*

2° Définition de la procédure *cercle* rendant pour résultat un module:

```
> cercle:=proc(c,r)
    module()
      export centre,rayon,diametre,aire,circonference;
      centre:=()->c;
      rayon:=()->r;
      diametre:=()->2*rayon();
      aire:=()->Pi*r^2;
      circonference:=()->Pi*diametre();
    end module
  end proc;
```

cercle := **proc**(*c*, *r*)
  module()
  export *centre*, *rayon*, *diametre*, *aire*, *circonference*;
    centre := ( ) → *c*;
    rayon := ( ) → *r*;
    diametre := ( ) → 2∗*rayon*( );
    aire := ( ) → π∗*r*^2;
    circonference := ( ) → π∗*diametre*( )
  end module
end proc

Définition d'un cercle *C* de centre *A* de rayon *R*:

```
> C:=cercle(A,R);
```

C := **module**() **export** *centre*, *rayon*, *diametre*, *aire*, *circonference*; **end module**

```
> C:-centre(), C:-rayon(), C:-diametre(), C:-aire(),
  C:-circonference();
```

$$A, R, 2\,R, \pi\,R^2, 2\,\pi\,R$$

Utilisation des deux structures pour obtenir les coordonnées du centre de *C*:

```
> C:-centre():-abscisse(), C:-centre():-ordonnee();
```

$$a, b$$

Travail dirigé 3:

TD 3.1:
Écrire une procédure ***second_degré(a,b,c)*** qui résout l'équation du second degré à coefficients réels: $a\,x^2 + b\,x + c = 0$ en distinguant 3 cas selon le signe du discriminant.

TD 3.2:
Écrire une procédure ***somme(n)*** calculant récursivement la somme des entiers de *0* à ***n*** , pour ***n*** entier naturel donné.

TD 3.3:
Écrire une procédure ***renverser(n)*** , qui , étant donné un entier naturel ***n*** ne comportant pas de 0 dans son écriture décimale , rend pour résultat l'écriture renversée de cet entier .
Prévoir un message d'erreur si l'entier comporte le chiffre 0.

Exemple:

```
> renverser(122564);
```

465221

```
> renverser(120325);
Error, (in renverser) le chiffre 0 n'est pas autorisé.
```

TD 3.4:
Écrire un module *segment* permettant de modéliser un segment du plan, défini par ses deux points extrémités.Ce module comportera des variables locales exportées rendant comme résultat la première et la seconde extrémité du segment, sa longueur, et son milieu.
On pourra utiliser le module défini par la procédure *pnt* de l'exercice corrigé **3.2**.