Aide-mémoire Ada (1)

Algorithmique, premier semestre

Domaine mathématiques et algorithmique

INSA 1ère année

NOM :

.....

PRÉNOM :

.....

GROUPE :

.....

D. LE BOTLAN

contact.lebotlan@insa-toulouse.fr

wwwperso.insa-toulouse.fr/~lebotlan/

Ce polycopié est le seul document autorisé pendant les contrôles notés. Il peut être annoté profusément, mais sans ajout de feuille.

N'hésitez pas à me contacter ou à demander des explications à vos encadrants si un point vous semble obscur. POSEZ DES QUESTIONS !

Table des matières

Identificateurs

— **Un identificateur** est un nom qui sert à repérer une entité du programme (telle qu'un sous-programme, un type, une variable, etc.). En Ada, les identificateurs ne doivent pas comporter d'espace, et on évitera les accents.

Exemples d'identificateurs : Nombre_Mots, Duree_Temporisation, Prenom_Client.

— **Foo**, **Bar**, **Moo**, . . . ne signifient rien de spécial, ils sont utilisés dans ce document lorsqu'il y a besoin d'un identificateur.

(On pourrait les remplacer par "Toto", "Lady_Gaga", ou juste "Z").

Commentaires

— **Un commentaire** dans un programme est un morceau qui n'est pas compilé (et qui est donc ignoré par l'ordinateur). En Ada, un commentaire commence par deux tirets, par exemple :

```
-- VOLUME DU MOTEUR EN CM3
Cylindree : constant Integer := 2450 ;
```

Blocs

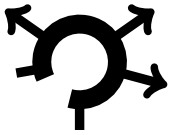
— **Un bloc** sert à décrire le comportement du programme complet ou d'un morceau de programme (on peut aussi dire « *bloc de code* »). Un bloc peut être soit :

— **Un bloc composé**, obtenu par la composition de blocs plus petits
(voir les différents types de blocs composés, pages 24, 26, et 28) ;

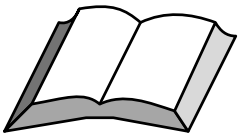
— **Une instruction élémentaire** : appel de procédure (voir pages 6 et 18)
ou une affectation de variable (voir page 12).

☞ **RÈGLE : Un bloc se termine toujours par un point-virgule.**

☞ Un programme est décrit principalement par son *comportement* et par les *données* qu'il manipule. Les symboles ci-dessous (rond-point et livre) sont utilisés pour représenter visuellement ces deux concepts.

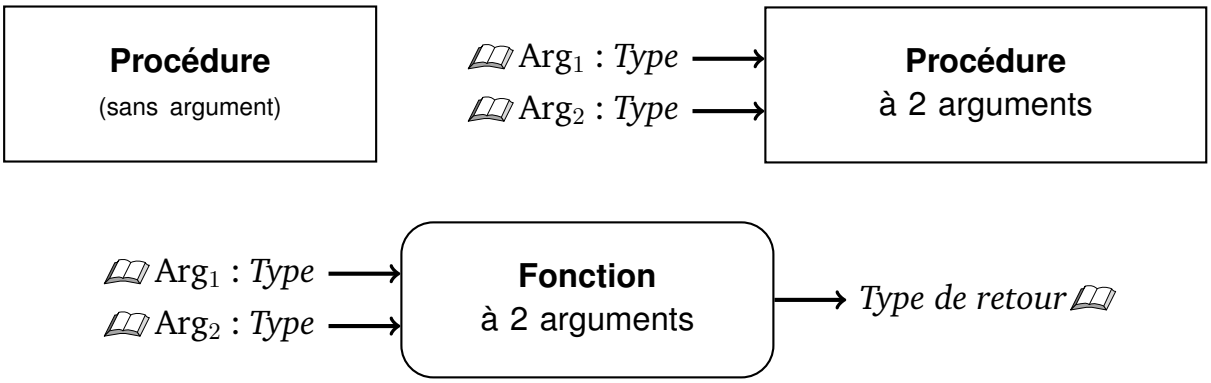


Comportement



Données

☞ Les symboles suivants représentent différentes sortes de sous-programmes :



☞ L'écriture d'un programme correct nécessite de connaître rigoureusement le *type* des données manipulées. Dans ce cours, on utilise des « jugements » qui peuvent être de plusieurs formes :

📖	$x \in \textit{type}$	pour indiquer le type d'une expression x
	$x \in \mathfrak{definition}$	pour indiquer que x est une définition.
🔄	$x \in \mathfrak{bloc}$	pour indiquer que x est un bloc de code.

Voici quelques exemples :

Jugement	*Interprétation*
$120 + 99 \in \textit{Integer}$	120 + 99 est de type « *Integer* »
$\mathsf{true} \in \textit{Boolean}$	true est de type « *Boolean* »
$\mathsf{foo} : \textit{Integer} \in \mathfrak{definition}$	foo : *Integer* est une définition (de la variable foo)
$\mathsf{foo} \in \textit{Integer}$	La variable foo est de type « *Integer* »
$\mathsf{foo} > 42 \in \textit{Boolean}$	foo > 42 est de type « *Boolean* »
$\mathsf{foo} := 42\,; \in \mathfrak{bloc}$	foo := 42 ; est un bloc de code

Pour alléger la notation, on écrit $\mathsf{foo} := 42 \in \mathfrak{bloc}$ (sans point-virgule).

Une expression n'est pas un bloc (et inversement) : $120 + 99 \notin \mathfrak{bloc}$

Structure d'un programme Ada

Il existe deux modèles de structures différents, selon que l'on veut produire un **programme exécutable** (à gauche) ou un **acteur** (à droite).

EXÉCUTABLE "mission-exe"

Un seul fichier : mission.adb

Fichier mission.adb

```
-- Déclaration du ou des acteurs utilisés
with Foo ;
procedure Mission is
   -- Renommage éventuel des acteurs
   package F renames Foo ;
   ...
   -- Définitions
   D_i ;
   ...
begin
   -- Corps du programme principal
   B ;
end Mission ;
```

Le corps du programme principal, $\mathcal{B}$, doit être un bloc, c.-à-d. $\mathcal{B} \in \mathfrak{bloc}$

La partie **avant le begin** est réservée aux définitions : $D_i \in \mathfrak{definition}$

ACTEUR "Foo"

Deux fichiers : foo.ads et foo.adb

Fichier foo.ads *(la spécification)*

```
-- Déclaration des acteurs utilisés
with Acteur_1 , Acteur_2 ;
package Foo is
   -- Renommage éventuel des acteurs
   package Bar renames Acteur_i ;
   ...
   -- Déclarations de sous-programmes
   ...
end Foo ;
```

⋆ La *déclaration* d'un sous-programme ne comprend pas le corps du sous-programme.

⋆ Pas de corps de programme (pas de **begin**), car un acteur n'est pas exécutable

Fichier foo.adb *(l'implémentation)*

```
with Acteur_1 , Acteur_2 ;
package body Foo is
   -- Définitions des sous-programmes
   ...
end Foo ;
```

Exemple : le fichier mission1.adb

```
1  with MarsRover ;
3  procedure Mission1 is
5      package MR renames MarsRover ;
7  begin
8      -- Invocation de l'action 'Avancer'
9      MR.Avancer ;
11     MR.Tourner_Droite ;
12 end Mission1 ;
```

☞ Se compile en l'exécutable "mission1-exe"

— Le corps de mission1.adb, lignes 8 à 11, est un bloc séquence (deux appels de procédure).

— L'**indentation** est le décalage à droite de certaines lignes du programme (p. ex., lignes 5, 8, 9, et 11 de mission1.adb).

Elle rend le code plus lisible en mettant en évidence les différentes parties (avant le **begin**, après le **begin**, les blocs).

Notes personnelles

Procédures sans argument

Une **PROCÉDURE** (appelée aussi “action”) est un **bloc** auquel on donne un nom.

Pour **INVOQUER** (c.-à-d. exécuter) la procédure **Bar**, on écrit

— `Moo.Bar ;` si la procédure Bar est définie dans l’acteur Moo.

— Juste `Bar ;` si la procédure Bar est définie dans le programme mission.adb.

Pour **DÉFINIR** une procédure dans le programme mission.adb :

Définition de procédure sans argument

```
procedure Bar is
   -- Définitions éventuelles
begin
   -- Corps de la procédure
   B
end Bar ;
```

∈ definition

À placer avant le **begin** du programme.

On doit avoir $\mathcal{B} \in$ bloc

RÈGLE *« Invocation »*

Une invocation de procédure est un bloc :

`Moo.Bar` ∈ bloc

`Bar` ∈ bloc

Exemple : le fichier mission2.adb

```
1  with GAda.Text_IO ;
3  procedure Mission2 is
5      package Txt renames GAda.Text_IO ;
7      procedure Afficher_Bienvenue is
8      begin
9          Txt.Put_Line(Aff => "Bonjour, ") ;
10         Txt.Put(Aff => "et bienvenue a l'INSA de ") ;
11     end Afficher_Bienvenue ;
13 begin
14     Afficher_Bienvenue ;
15     Txt.Put_Line(Aff => "Toulouse") ;
17     Afficher_Bienvenue ;
18     Txt.Put_Line(Aff => "Rennes") ;
20     Afficher_Bienvenue ;
21     Txt.Put_Line(Aff => "Lyon") ;
22 end Mission2 ;
```

Définition de la procédure Afficher_Bienvenue ∈ definition

Afficher_Bienvenue

Invocations de la procédure ∈ bloc

- ☞ Ce programme affiche trois fois “Bonjour, et bienvenue ...”
- ☞ Il contient trois invocations de la procédure Afficher_Bienvenue (lignes 14, 17, et 20) et cinq invocations d’actions contenues dans l’acteur GAda.Text_IO (lignes 9, 10, 15, 18, et 21).
- ☞ La procédure Afficher_Bienvenue est **définie** entre les lignes 7 et 11. Les procédures Put et Put_Line sont définies dans l’acteur GAda.Text_IO.

Notes personnelles

Bloc Séquence

Si B_1, B_2, .., B_n sont des blocs de code, leur juxtaposition constitue un nouveau bloc appelé BLOC SÉQUENCE. L'exécution du bloc séquence consiste à exécuter chaque bloc B_1, B_2, .., B_n, l'un après l'autre :

Bloc séquence

La séquence des blocs B_1, .., B_n s'écrit

$$\begin{array}{|l} B_1\ ; \\ B_2\ ; \\ \vdots \\ B_n\ ; \end{array}$$

- ☞ Le bloc B_1 est exécuté en premier.
- ☞ Le bloc B_2 ne sera exécuté que lorsque le bloc B_1 sera **terminé**.
- ☞ Le bloc séquence est terminé lorsque le dernier bloc, B_n, est terminé.

Le corps du programme ci-dessous est un bloc séquence.

Exemple : séquence d'actions

```
with Moteur_Hybride ;
procedure Mission1 is
  package MH renames Moteur_Hybride ;
begin
   MH.Demarrage_Electrique ;
   MH.Demarrage_Thermique ;
   MH.Laisser_Tourner ;
   MH.Arreter ;
end Mission1 ;
```

$\in$ bloc

RÈGLE « *Séquence* »

Si pour tout $1 \leqslant i \leqslant n$, $\mathcal{B}_i$ est un bloc, alors leur juxtaposition est un bloc :

$$\forall i \in [1;n] \quad \mathcal{B}_i \in \text{bloc} \quad \Rightarrow \quad \begin{array}{|l} \mathcal{B}_1\ ; \\ \vdots \\ \mathcal{B}_n\ ; \end{array} \in \text{bloc}$$

La séquence vide s'écrit null et est un bloc : null $\in$ bloc.

Les types de base

Un programme a vocation à manipuler des données (on peut aussi dire des « *valeurs* »), p. ex., pour effectuer des calculs numériques. Afin de manipuler correctement chaque valeur, l'ordinateur doit savoir à quelle catégorie ou **type** elle appartient. Par défaut, Ada propose les types de base suivants :

Nom du type	Signification	Valeurs	Place mémoire
INTEGER	entiers	de $-2\_147\_483\_648$ à $+2\_147\_483\_647$	4 octets (32 bits)*
NATURAL	entiers naturels	de 0 à $+2\_147\_483\_647$	*idem*
POSITIVE	entiers positifs	de 1 à $+2\_147\_483\_647$	*idem*
FLOAT	nombres réels	de $-3.4E^{+38}$ à $3.4E^{+38}$	4 octets (32 bits)^
BOOLEAN	booléens	True (vrai), False (faux)	1 octet
CHARACTER	caractères	'0', '9', 'A', 'Z', 'a', 'z', '=', ...	1 octet
STRING	chaînes de caractères	`"Du texte."` (pour une chaîne de n caractères)	n octets

*Sur les machines 64 bits, un entier occupe 8 octets et va jusqu'à 9.10^{18}

^Un réel peut aussi occuper 64 bits (y compris sur des machines 32 bits) et aller jusqu'à 10^{308}.

RÈGLES *« Types de base »*

-2000 ∈ *Integer*
42 ∈ *Integer*
42 ∈ *Natural*
42 ∈ *Positive*
42.0 ∈ *Float*
True ∈ *Boolean*
False ∈ *Boolean*
'A' ∈ *Character*
'8' ∈ *Character*
"Moo" ∈ *String*
"42" ∈ *String*

Les constantes

On s'interdira d'utiliser des valeurs numériques dans le corps du programme (p. ex., 3.141593) ; on utilisera à la place des **constantes**.

Définition d'une constante

Foo : **constant** son_type := sa_valeur ; $\in$ definition

À placer avant le **begin** du programme.

RÈGLE *« Constante »*

$e \in$ *un_type* $\Rightarrow$ Bar : **constant** *un_type* := e $\in$ definition

Après la définition de la constante Bar, le jugement Bar $\in$ *un_type* est valide.

Exemples de définitions de constantes

```
-- Ces définitions de constantes sont placées avant le begin du programme
Pi                          : constant Float   := 3.1415927 ;
-- Vitesse en m.s^-2
V_Lumiere                   : constant Float   := 3.0E8   ;
-- Vitesse en tours par minute
V_Rotation_Max              : constant Float   := 8200.0  ;

Nombre_Eleves               : constant Integer := 462 ;
Annee_Accession_Akhenaton   : constant Integer := -1_348 ;

Nom_Appareil                : constant String  := "Airbus A380" ;

Est_En_Mode_Simulation      : constant Boolean := False ;
```

RÈGLE : un nombre réel (**Float**) contient **toujours** un point décimal.

En application de la règle « Constante », on obtient :

Pi $\in$ *Float*
V_Lumiere $\in$ *Float*
V_Rotation_Max $\in$ *Float*
Nombre_Eleves $\in$ *Integer*
Nom_Appareil $\in$ *String*

Une variable est semblable à une constante que l'on pourrait modifier pendant l'exécution du programme.

Variable de type *Integer*

· Pour **DÉFINIR** une variable Foo de type *Integer*, il suffit de placer `Foo : Integer ;` (definition) avant le **begin** du sous-programme.

· Une **AFFECTATION** permet de modifier la valeur de la variable : `Foo := expression numérique ;` (bloc) p. ex. `Foo := 60 * 24 ;`

· Pour **UTILISER** la valeur de la variable (dans une expression numérique), il suffit d'écrire son nom, par exemple `12 + Foo * 64` (*Integer*).

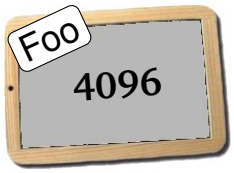
RÈGLE « *Variable* »

Il existe deux manières de définir une variable (on suppose $e \in$ *Integer*) :

`Foo : Integer` $\in$ definition (simple déclaration)

`Foo : Integer := e` $\in$ definition (déclaration + affectation)

Après la définition de la variable Foo, Foo $\in$ *Integer* est valide.



`Foo := 4096` $\in$ bloc

RÈGLE « *Affectation* »

Foo $\in$ *Integer* et $e \in$ *Integer* $\Rightarrow$ `Foo := e` $\in$ bloc

(c.-à-d., une affectation est un **bloc**)

Variables d'autres types

Les variables peuvent être de n'importe quel type, en particulier :

— `Foo : Float` ($\in$ definition) définit une **VARIABLE RÉELLE**.

— `Foo : Boolean` ($\in$ definition) définit une **VARIABLE BOOLÉENNE**.

✖ **Les variables de type *String*** sont interdites (uniquement des constantes).

Exemple d'utilisation de variables et de constantes

```
   ⋮
   -- Seuil de résistance du système à l'accélération (3g)
   Seuil_Resistance : constant Float := 3.0 * 9.81 ;          ∈ definition
   -- Précision relative de la mesure de l'accélération (4%)
   Precision_Mesure : constant Float := 0.04 ;                ∈ definition

   Acceleration : Float ;                                     } Définition de deux variables
   Acceleration_Est_Acceptable : Boolean ;                      ∈ definition
begin
   -- On suppose que la fonction Mesurer_Acceleration est définie (elle renvoie un réel).
   Acceleration := Mesurer_Acceleration ;

   -- Majoration de la mesure avec la précision relative
   Acceleration := Acceleration * (1.0 + Precision_Mesure) ;

   Acceleration_Est_Acceptable := Acceleration < Seuil_Resistance ;
```

Trois affectations
∈ bloc

Expression numérique

Un calcul s'exprime à l'aide d'une **expression numérique**, comme celle que l'on utilise sur une calculatrice. Par exemple :

```
Nb_Secondes_dans_Annee : constant Integer := 365 * 24 * 3600 ;
```

☞ Une expression numérique doit être **homogène**, c.-à-d. uniquement additionner, multiplier, diviser, etc., des entiers entre eux ou des réels entre eux. Cependant, il est possible d'utiliser une conversion (voir ci-dessous).

☞ Une expression numérique peut utiliser des constantes, des variables ou des fonctions si elles sont du même type. (Voir les exemples page ci-contre.)

Classification

$e \in$ *Integer*	expression entière
$e \in$ *Float*	expression réelle
$e \in$ *Boolean*	expression booléenne (= assertion)

RÈGLE *« Opérateurs arithmétiques »*

Si e et e' sont deux expressions telles que $e \in$ *Integer* et $e' \in$ *Integer*, alors

$e + e'$	$\in$ *Integer*	
$e - e'$	$\in$ *Integer*	
$e * e'$	$\in$ *Integer*	
e / e'	$\in$ *Integer*	division entière
e mod e'	$\in$ *Integer*	modulo
abs(e)	$\in$ *Integer*	valeur absolue

Idem pour les opérateurs sur les *Float*.

Conversion

— Réel vers entier (Float vers Integer), par arrondi : `Integer (valeur_réelle)`

— Entier vers réel (Integer vers Float), par injection : `Float (valeur_entière)`

Ainsi, Integer(12.5) vaut 13 et Float(132) vaut 132.0.

RÈGLE *« Conversion »*

$e \in$ *Integer* $\Rightarrow$ Float $(e) \in$ *Float*
$e \in$ *Float* $\Rightarrow$ Integer $(e) \in$ *Integer*

RÈGLE *« Puissance »*

La puissance n^{eme}, x^n, s'écrit `x ** n`

$$\begin{array}{l} e \in \textit{Float} \\ \text{et } e' \in \textit{Integer} \end{array} \Rightarrow e ** e' \in \textit{Float}$$

Exemples d'expressions numériques

☞ Une expression numérique peut utiliser des constantes ou des variables (comme ici Vitesse $\in$ *Float* et Jour_Semaine $\in$ *Integer*) :

(Jour_Semaine + 1) mod 7 $\in$ *Integer*

(35.0 * 35.0) / (50.5 - Vitesse) $\in$ *Float*

mod est l'opérateur « modulo » x mod y renvoie le reste de la division entière de x par y.

☞ Il est aussi possible d'utiliser des fonctions :

50 - Foo(x => 34) * Foo(x => 11 / 2) $\in$ *Integer*

Foo est une fonction définie avant le **begin** du programme

(Voir la section sur les fonctions, page 20)

Exemples de calculs utilisant des conversions

```
-- Ces définitions de constantes et variables sont placées avant le begin
Frequence      : Integer := 135E6 ;                    -- (soit 135 MHz)
Periode        : Float   := 1.0 / Float(Frequence) ;
Periode_Fausse : Float   := Float( 1 / Frequence ) ;   -- Ce calcul renvoie zéro !
Demi_Periode   : Float   := Periode / 2.0 ;            -- On doit diviser un réel par un réel.
```

Notes personnelles

Expression booléenne (assertions)

Une expression booléenne (ou assertion) est une expression e telle que $e \in$ *Boolean*. Elle peut être ou bien vraie (True) ou bien fausse (False).

Les expressions booléennes sont utiles dans un bloc **IF** ou un bloc **WHILE**

(voir pages 24 et 28)

Opérateurs booléens à connaître

Opérateur	Évaluation
OR Ou logique	a **or** b est vraie si a est vraie **ou** b est vraie.
AND Et logique	a **and** b est vraie si a est vraie **et** b est vraie.
NOT Non logique	**not** a vaut le **contraire** de a.

RÈGLE *« Opérateurs booléens »*

$e \in$ *Boolean* et $e' \in$ *Boolean*

$\Rightarrow$ e **or** e' $\in$ *Boolean*

$\Rightarrow$ e **and** e' $\in$ *Boolean*

$\Rightarrow$ **not** e $\in$ *Boolean*

Opérateurs de comparaison sur les nombres

Opérateur	Signification
<	Inférieur strictement à
<=	Inférieur ou égal à
>	Supérieur strictement à
>=	Supérieur ou égal à
=	Égal à
/=	Différent de

RÈGLE *« Comparaisons »*

$e \in$ *Integer* et $e' \in$ *Integer*

$\Rightarrow$ e **<** e' $\in$ *Boolean*

$\Rightarrow$ e **<=** e' $\in$ *Boolean*

$\Rightarrow$ e **>** e' $\in$ *Boolean*

$\Rightarrow$ e **>=** e' $\in$ *Boolean*

$\Rightarrow$ e **=** e' $\in$ *Boolean*

$\Rightarrow$ e **/=** e' $\in$ *Boolean*

☞ Ces opérateurs de comparaison s'appliquent aussi à deux réels (*Float*), deux chaînes de caractères (*String*), et à d'autres types (p. ex., type article).

RÈGLE : On n'utilise jamais l'égalité pour comparer deux **réels**.

(Ni l'opérateur « différent de »)

Notamment, `if x = 0.0 then ...` et `if x /= 0.0 then ...` sont interdits.

Au lieu d'utiliser le test d'égalité, on compare la distance entre les deux nombres avec un seuil ϵ : ainsi, au lieu de `x = 0.0`, on écrit plutôt `if abs(x) < epsilon then ...`, où `epsilon` est une constante définie au début du programme.

Définition de procédure avec arguments

Le corps d'une procédure avec arguments dépend de paramètres, que l'on appelle « **arguments formels** » (ou « paramètres formels »).

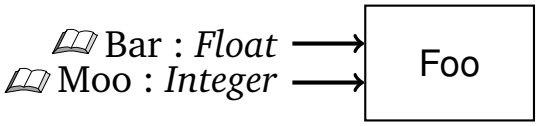
Définition de procédure à deux arguments (exemple)

```
procedure Foo (Bar : Float ; Moo : Integer) is
   -- Définitions éventuelles
begin
   -- Corps de la procédure, utilise Bar et Moo
   B ;
end Foo ;
```

∈ definition

À placer avant le **begin**.

$\mathcal{B}$ doit être un bloc :
$\mathcal{B} \in$ bloc



Invocation de procédure avec arguments

La procédure Foo est déjà définie, soit dans le programme, soit dans un acteur.

L'**invocation** de la procédure Foo se fait en fournissant les « **arguments d'appel** » (ou « paramètres d'appel ») qui doivent être du bon type. Les trois invocations suivantes sont équivalentes :

```
Foo (Bar => 4.5, Moo => 120) ;   ∈ bloc
Foo (Moo => 120, Bar => 4.5) ;   ∈ bloc
Foo (4.5, 120) ;                 ∈ bloc
```

Arguments d'appel

RÈGLE *« Appel de procédure »*

Un appel de procédure est un bloc (exemple) :

$e \in Float$ et $e' \in Integer$ $\Rightarrow$ Foo (Bar => e, Moo => e') $\in$ bloc

Exemple de procédure avec argument

```
with GAda.Text_IO ;
procedure Mission2 is
   package Txt renames GAda.Text_IO ;
   -- CETTE PROCÉDURE AFFICHE UN MESSAGE DE BIENVENUE PARAMÉTRÉ
   procedure Afficher_Bienvenue (Nom_INSA : String) is
   begin
      Txt.Put_Line (Aff => "Bonjour, ") ;
      Txt.Put_Line (Aff => "bienvenue a l'INSA de " & Nom_INSA) ;
   end Afficher_Bienvenue ;
begin
   Afficher_Bienvenue (Nom_INSA => "Toulouse") ;
   Afficher_Bienvenue (Nom_INSA => "Rennes") ;
   Afficher_Bienvenue (Nom_INSA => "Lyon") ;
end Mission2 ;
```

∈ definition

& permet de coller deux chaînes

∈ bloc (bloc séquence)

📖 Nom_INSA : *String* ⟶ Afficher_Bienvenue

Note : cette procédure **AFFICHE** un message mais ne renvoie aucun résultat (au contraire d'une **FONCTION**).

Définition de fonction avec arguments

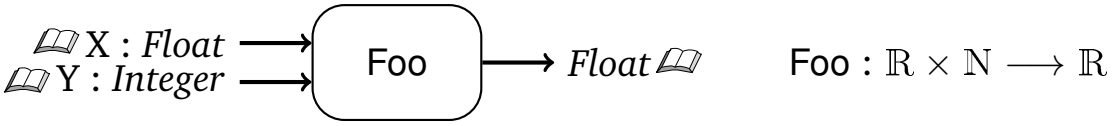
Une fonction **RENVOIE** un résultat (une valeur), alors qu'une procédure ne renvoie rien.

Définition de fonction à deux arguments (exemple)

```
function Foo (X : Float ; Y : Integer) return Float is
   -- Définitions éventuelles
   Resultat : Float ;
begin
   -- Corps de la fonction B ∈ bloc
   B ;
   -- Se termine forcément par return
   return Resultat ;
end Foo ;
```

La variable Resultat est calculée dans le corps de la fonction.

∈ definition



Invocation de fonction avec arguments

La fonction Foo est déjà définie, soit dans le programme, soit dans un acteur.

L'**invocation** de la fonction Foo se fait en fournissant les arguments (ici X et Y) qui doivent être du bon type. Les trois invocations suivantes sont équivalentes :

```
Foo (X => 4.5, Y => 120)   ∈ Float
Foo (Y => 120, X => 4.5)   ∈ Float
Foo (4.5, 120)             ∈ Float
```

Une invocation de fonction n'est pas un bloc ! On ne peut pas écrire :

~~**begin**~~
~~Foo (X => 4.5, Y => 120) ;~~

RÈGLE *des* **Return**

Le premier **return** est suivi d'un type :

function Foo (...) **return** *un_type* **is**

Le second **return** est suivi d'une expression de ce type :

return e ; avec $e \in$ *un_type*

RÈGLE *« Appel de fonction »*

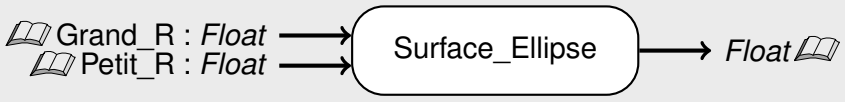
Un appel de fonction est une expression (ce n'est pas un bloc) :

$e \in$ *Float* et $e' \in$ *Integer*

$\Rightarrow$ Foo (X => e, Y => e') $\in$ *Float*

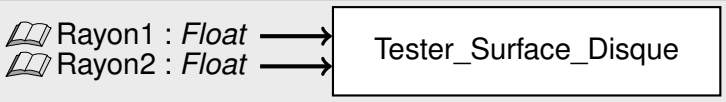
Exemple complet : le fichier mission.adb

```
1  with GAda.Text_IO ;
3  procedure Mission is
5     package Txt renames GAda.Text_IO ;
7     Pi : constant Float := 3.1415927 ;
9     -- DÉFINITION DE LA FONCTION Surface_Ellipse                    ∈ definition
10    function Surface_Ellipse (Grand_R : Float ; Petit_R : Float) return Float is
11    begin
12       return (Pi * Grand_R * Petit_R) ;
13    end Surface_Ellipse ;
```



Cette fonction est invoquée ici

```
16    -- PROCÉDURE DE TEST DE LA FONCTION                            ∈ definition
17    procedure Tester_Surface_Ellipse (Rayon1 : Float ; Rayon2 : Float) is
18       Aire : Float ;
19    begin
20       Txt.Put (Aff => "Une ellipse de rayons " & Float'Image(Rayon1)
21                & " et " & Float'Image(Rayon2)) ;
23       Aire := Surface_Ellipse (Grand_R => Rayon1, Petit_R => Rayon2) ;
24       Txt.Put_Line (Aff => " a pour surface " & Float'Image(Aire)) ;
25    end Tester_Surface_Ellipse ;
```




```
27 begin
28    Tester_Surface_Ellipse (Rayon1 => 1.0, Rayon2 => 1.0) ;
29    Tester_Surface_Ellipse (Rayon1 => 2.0, Rayon2 => 3.0) ;
30    Tester_Surface_Ellipse (Rayon1 => 3.0, Rayon2 => 4.0) ;
31 end Mission ;
```

Trois appels de la procédure (∈ bloc)

Ce programme, lorsqu'il est compilé puis exécuté, affiche ceci à l'écran :

```
Une ellipse de rayons 1.000E+00 et 1.000E+00 a pour surface 3.14159E+00
Une ellipse de rayons 2.000E+00 et 3.000E+00 a pour surface 1.88496E+01
Une ellipse de rayons 3.000E+00 et 4.000E+00 a pour surface 3.76991E+01
```

Noter que la procédure Tester_Surface_Ellipse ne *renvoie* aucune valeur (car ce n'est pas une fonction), mais a un *effet* : elle **affiche** un message à l'écran.

```
Surface_Ellipse (Grand_R => 10.0, Petit_R => 10.0)   ∈ Float
Tester_Surface_Ellipse (Rayon1 => 5.0, Rayon2 => 3.0)   ∈ bloc
```

Types ARTICLE

Un **type article** est l'agrégation de plusieurs types en un seul type.

Définition d'un type article

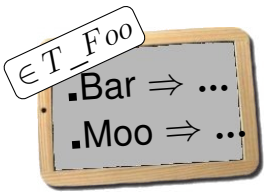
```
type T_Foo is record
   Bar : un_type1 ;
    ⋮
   Moo : un_typen ;
end record ;
```

∈ definition

Accès aux attributs

Si Zoo est une variable de type T_Foo, l'expression `Zoo.Bar` renvoie la valeur de l'attribut Bar.

Pour modifier la valeur de l'attribut, utiliser `Zoo.Bar := valeur ;`



RÈGLE *« Accès aux attributs »* 

On suppose que le type article T_Foo contient un attribut Bar de type un_type_1 (comme dans la définition ci-dessus).

$e \in T\_Foo \quad \Rightarrow \quad$ `e . Bar` $\in un\_type_1$

$e \in T\_Foo$ et $e' \in un\_type_1 \quad \Rightarrow \quad$ `e . Bar := e'` $\in$ bloc

De même pour les autres attributs (Moo, etc.)

RÈGLE *« Construction d'un article »*

On suppose que le type T_Foo est défini comme ci-dessus.

$e_1 \in un\_type_1 \quad \ldots \quad e_n \in un\_type_n$

$\Rightarrow$ $(e_1, \ldots, e_n) \in T\_Foo$

$\Rightarrow$ (Bar => e_1, ..., Moo => e_n) $\in T\_Foo$

$\Rightarrow$ (Moo => e_n, ..., Bar => e_1) $\in T\_Foo$

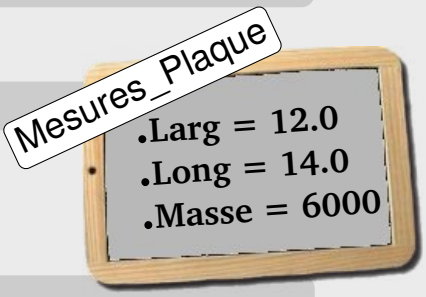
Exemple de type article

```
procedure Mission is
   type T_Mesures is record
      Larg   : Float ;
      Long   : Float ;
      Masse  : Integer ;
   end record ;                                   ∈ definition

   -- Fonction calculant la densité Surfacique   ∈ definition
   function Densite (Mes : T_Mesures) return Float is
   begin
      return Float (Mes.Masse) / (Mes.Larg * Mes.Long) ;
   end Densite ;

   -- Définition d'une variable de type T_Mesures   ∈ definition
   Mesures_Plaque : T_Mesures ;
begin
   -- Modification des attributs, un par un
   Mesures_Plaque.Larg  := 12.0 ;
   Mesures_Plaque.Long  := 14.0 ;
   Mesures_Plaque.Masse := 6000 ;
   -- Modification de tous les attributs
   Mesures_Plaque := (12.0, 14.0, 6000) ;
   -- Modification de tous les attributs
   Mesures_Plaque := (Larg => 12.0, Long => 14.0, Masse => 6000) ;
end Mission ;
```

Définition du type T_Mesures

Mes : *T_Mesures* → Densite → *Float*

(On ne s'en sert pas dans le corps du programme)



Ces trois manières de modifier les attributs sont équivalentes.

Bloc IF

Le bloc **IF** permet d'exécuter ou bien un bloc de code $\mathcal{B}$, ou bien un bloc de code alternatif $\mathcal{B}'$, selon qu'une condition donnée est vraie ou fausse.

Définition du bloc **IF**

Syntaxe :

if *condition* **then**
  $\mathcal{B}$;
else
  $\mathcal{B}'$;
end if ;

avec *condition* $\in$ *Boolean*
(« *condition* » est une expression booléenne, voir page 16)

$\mathcal{B} \in \mathfrak{bloc}$ et $\mathcal{B}' \in \mathfrak{bloc}$

Exécution du bloc **IF** :

1 – La condition est évaluée à vrai ou faux (True ou False) ;
2a – Si c'est *vrai*, le bloc $\mathcal{B}$ est exécuté (mais pas $\mathcal{B}'$)
2b – Si c'est *faux*, le bloc $\mathcal{B}'$ est exécuté (mais pas $\mathcal{B}$)
3 – Le bloc **IF** est terminé lorsque le bloc exécuté ($\mathcal{B}$ ou $\mathcal{B}'$) est terminé.

Lorsque le bloc $\mathcal{B}'$ est vide, on peut écrire : **if** *condition* **then** $\mathcal{B}$ **end if ;**

RÈGLE « *Bloc* **IF** »

Un bloc **IF** est un bloc :

$e \in$ *Boolean*, $\mathcal{B} \in \mathfrak{bloc}$ et $\mathcal{B}' \in \mathfrak{bloc}$,

$\Rightarrow$

if e **then**
  $\mathcal{B}$;
else
  $\mathcal{B}'$;
end if ;

$\in \mathfrak{bloc}$

Exemple de bloc **IF**

Noter qu'il n'est pas possible d'écrire la condition ~~90 < Poids <= 100~~.
À la place, on écrit :

```
if (90 < Poids) and (Poids <= 100) then
   Txt.Put(Aff => "Categorie Mi-Lourd") ;
else
   Txt.Put(Aff => "Autre categorie") ;
end if ;
```

∈ bloc

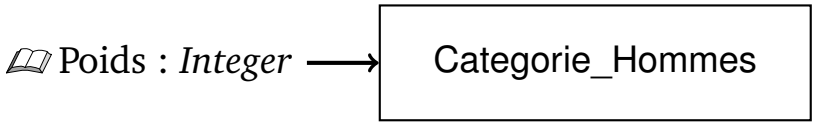
Dans un bloc **if**, le bloc $\mathcal{B}$ et le bloc $\mathcal{B}'$ sont des blocs quelconques. En particulier, $\mathcal{B}'$ peut être un bloc **if**, ce qui permet d'enchaîner les tests :

```
-- Cette procédure affiche à l'écran la catégorie (au judo) selon le poids.
procedure Categorie_Hommes (Poids : Integer) is
begin
   if Poids < 60 then
      Txt.Put_Line(Aff => "Super leger") ;
   elsif Poids <= 66 then
      Txt.Put_Line(Aff => "Mi-leger") ;
   elsif Poids <= 73 then
      Txt.Put_Line(Aff => "Leger") ;
   elsif Poids <= 81 then
      Txt.Put_Line(Aff => "Mi-moyen") ;
   elsif Poids <= 90 then
      Txt.Put_Line(Aff => "Moyen") ;
   elsif Poids <= 100 then
      Txt.Put_Line(Aff => "Mi-lourd") ;
   else
      Txt.Put_Line(Aff => "Lourd") ;
   end if ;
end Categorie_Hommes ;
```

∈ definition

Poids	Catégorie
< 60kg	Super léger
60kg – 66kg	Mi-léger
66kg – 73kg	Léger
73kg – 81kg	Mi-moyen
81kg – 90kg	Moyen
90kg – 100kg	Mi-lourd
> 100kg	Lourd

Noter que l'on écrit **elsif** au lieu de **else if**
(Cette écriture **elsif** permet aussi de n'écrire qu'un seul **end if** à la fin du bloc, au lieu de six).



Bloc FOR

Prenons les trois ingrédients d'un bloc **FOR** :

— x est un identificateur, p. ex. `Numero_d_Electrovanne`

— $\mathcal{B}(x)$ est un bloc, par exemple $\mathcal{B}(x) =$ `Fermer_Electrovanne (x);`

— a .. b est un intervalle de $\mathbb{Z}$: p. ex. `1 .. 12`

Le bloc FOR exécute $\mathcal{B}(x)$ pour chaque x de l'intervalle $[a..b]$.

Définition du bloc **FOR**

Syntaxe :

```
for x in a .. b loop
   B(x);
end loop;
```

☞ x est un identificateur quelconque
☞ $a \in$ *Integer* et $b \in$ *Integer*
☞ $\mathcal{B}(x) \in$ bloc

Le bloc **FOR** est équivalent à la séquence $\mathcal{B}(a)\,;\ \mathcal{B}(a+1)\,;\ \ldots\,;\ \mathcal{B}(b)\,;$

RÈGLE *« Bloc* **FOR** *»*

Un bloc **FOR** est un bloc :

$a \in$ *Integer*, $b \in$ *Integer* et $\mathcal{B}(x) \in$ bloc,

$\Rightarrow$

```
for x in a .. b loop
   B(x);
end loop;
```

$\in$ bloc

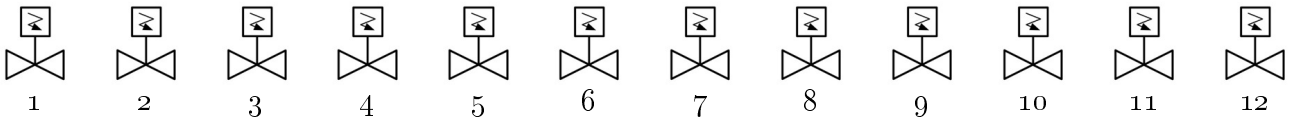
Exemple de bloc **FOR**

```
-- Ferme les 12 électrovannes numérotées de 1 à 12.
Nb_Electrovannes : constant Integer := 12 ; (∈ definition)
begin
   for Numero_Electrovanne in 1..Nb_Electrovannes loop
      Fermer_Electrovanne (Num => Numero_Electrovanne) ;
   end loop ;
```

Équivalent au bloc séquence :

```
Fermer_Electrovanne(Num => 1) ;
Fermer_Electrovanne(Num => 2) ;
      ⋮
Fermer_Electrovanne(Num => 12) ;
```

∈ bloc



Le bloc **FOR** est aussi utile pour répéter une action un certain nombre de fois. Par exemple :

```
-- CE BLOC FOR ÉMET 58 FOIS LE MÊME MESSAGE.
Nb_Repetitions : constant Integer := 58 ; (∈ definition)
begin
for Numero in 1..Nb_Repetitions loop
   Emettre_Message (Msg => "What hath God wrought?") ;
end loop ;
```

Noter que Numero n'est pas utilisé dans le corps du bloc **for**.

Le bloc $\mathcal{B}(x)$ est quelconque, en particulier cela peut être un bloc **for** imbriqué :

Exemple de blocs **FOR** *imbriqués*
Le fichier mission2.adb *:*

```
with GAda.Text_IO ;
procedure Mission2 is
   package Txt renames GAda.Text_IO ;
   Fin : constant Integer := 6 ;
begin
   for N_Ligne in 1..Fin loop
      for N_Colonne in 1..N_Ligne loop
         Txt.Put(Aff => Integer'Image(N_Ligne) ) ;
         Txt.Put(Aff => Integer'Image(N_Colonne) ) ;
         Txt.Put(Aff => " -- " ) ;
      end loop ;
      Txt.New_Line ;
   end loop ;
end Mission2 ;
```

L'exécution de mission2-exe affiche ceci à l'écran :

```
1 1 --
2 1 -- 2 2 --
3 1 -- 3 2 -- 3 3 --
4 1 -- 4 2 -- 4 3 -- 4 4 --
5 1 -- 5 2 -- 5 3 -- 5 4 -- 5 5 --
6 1 -- 6 2 -- 6 3 -- 6 4 -- 6 5 -- 6 6 --
```

Bloc WHILE

Alors que le bloc **for** parcourt un intervalle d'entiers, le bloc **WHILE** répète un bloc $\mathcal{B}$ **tant qu'une condition est vérifiée** (c.-à-d., jusqu'à ce que la condition ne soit plus vérifiée).

Définition du bloc **WHILE**

Syntaxe :
while *condition* **loop**
  $\mathcal{B}$;
end loop ;

avec $condition \in Boolean$ (voir page 16)
et $\mathcal{B} \in \mathfrak{bloc}$

Exécution du bloc **WHILE** :

1 – La condition est évaluée à vrai, ou faux (True, ou False) ;
2a – Si c'est *vrai*, le bloc $\mathcal{B}$ est exécuté, puis le bloc **while** est exécuté de nouveau (retour à l'étape 1).
2b – Si c'est *faux*, le bloc **while** est terminé.

Invariant : Le bloc **while** ne termine que si la condition est fausse.
Corollaire : L'exécution du bloc $\mathcal{B}$ doit modifier la valeur de la condition*.

*Sinon le bloc **while** se répète sans arrêt et ne termine jamais.

Vocabulaire : chaque répétition du bloc while s'appelle une *itération*.

RÈGLE « *Bloc* **WHILE** »

Un bloc **WHILE** est un bloc :

$e \in Boolean$ et $\mathcal{B} \in \mathfrak{bloc}$,

$\Rightarrow$ **while** e **loop** $\mathcal{B}$; **end loop ;** $\in \mathfrak{bloc}$

Exemple de bloc **WHILE**

```
1   -- Niveau est une variable réelle déclarée avant le begin
2   -- et Mesurer_Niveau_Cuve une fonction sans argument

5   -- Mesure initiale
6   Niveau := Mesurer_Niveau_Cuve ;

8   Ouvrir_Electrovanne ;

10  while Niveau < Capacite loop
11      Niveau := Mesurer_Niveau_Cuve ;
12  end loop ;

14  Fermer_Electrovanne ;
```

∈ bloc

Tant que le niveau mesuré est inférieur à Capacite,

On met à jour la mesure du niveau (le remplissage continue puisque la vanne est ouverte).

Le bloc **while** se termine lorsque le niveau mesuré est supérieur ou égal à Capacite
La vanne est ensuite fermée (ligne 14) et le remplissage s'arrête.

☞ La mise à jour de la variable Niveau dans le bloc **while** (ligne 11) permet, au final, de modifier la valeur de la condition Niveau < Capacite, et de sortir du bloc.

☞ Cette mise à jour, ligne 11, est cruciale pour ne pas boucler indéfiniment (et ne pas faire déborder la cuve).

Les acteurs GAda.Text_IO

Un programme a parfois besoin d'écrire des informations à l'écran ou de demander des informations à l'utilisateur (p. ex., son nom ou sa date de naissance). On utilisera pour cela les acteurs suivants :

L'acteur GAda.Text_IO

```
-- PUT AFFICHE LE MESSAGE PASSÉ EN ARGUMENT, SANS PASSER À LA LIGNE
procedure Put (Aff : String) ;
-- PUT_LINE AFFICHE LE MESSAGE ET PASSE À LA LIGNE
procedure Put_Line (Aff : String) ;
-- NEW_LINE PASSE À LA LIGNE
procedure New_Line ;
-- FGET LIT UNE CHAÎNE DE CARACTÈRE AU CLAVIER
function FGet return String ;
```

L'acteur GAda.Integer_Text_IO

```
-- FGET LIT UN ENTIER AU CLAVIER
function FGet return Integer ;
```

L'acteur GAda.Float_Text_IO

```
-- FGET LIT UN RÉEL AU CLAVIER
function FGet return Float ;
```

Opérateur et fonctions sur les chaînes

— L'opérateur `&` permet de coller ensemble deux chaînes (*concaténation*)

— La fonction `Integer'Image ( ...)` transforme un entier en chaîne.

— La fonction `Float'Image ( ...)` transforme un réel en chaîne.

Voici un exemple typique d'utilisation (X étant une variable entière) :

```
GAda.Text_IO.Put (Aff => "La variable X vaut " & Integer'Image(X) )
```

RÈGLE *« Opérateurs et fonctions sur les chaînes »*

$e \in \mathit{String}$ et $e' \in \mathit{String} \Rightarrow$ `e & e'` $\in \mathit{String}$

$e \in \mathit{Integer} \Rightarrow$ `Integer'Image (e)` $\in \mathit{String}$

$e \in \mathit{Float} \Rightarrow$ `Float'Image (e)` $\in \mathit{String}$

Exemple d'utilisation des acteurs GAda.Text_IO

```
with GAda.Text_IO, GAda.Integer_Text_IO, GAda.Float_Text_IO ;
procedure Mission3 is
   package Txt  renames GAda.Text_IO ;
   Premier_Terme : Float ;
   Raison        : Float ;
   Nombre_Termes : Integer ;
   Terme_Courant : Float ;
begin
   -- INTRODUCTION
   Txt.Put_Line (Aff => "Bonjour, ce programme affiche " &
                        "les premiers termes d'une suite geometrique") ;
   Txt.New_Line ;
   -- LECTURE DE LA VALEUR DU 1ER TERME
   Txt.Put (Aff => "Quelle est la valeur du premier terme ? ") ;
   Premier_Terme := GAda.Float_Text_IO.FGet ;
   -- LECTURE DE LA VALEUR DE LA RAISON
   Txt.Put (Aff => "Quelle est la valeur de la raison ? ") ;
   Raison := GAda.Float_Text_IO.FGet ;
   -- LECTURE DU NOMBRE DE TERMES
   Txt.Put (Aff => "Combien de termes voulez-vous ? ") ;
   Nombre_Termes := GAda.Integer_Text_IO.FGet ;
   -- AFFICHAGE DES TERMES
   -- On part du 1er terme
   Terme_Courant := Premier_Terme ;
   for No_Terme in 1 .. Nombre_Termes loop
      Txt.Put_Line (Aff => "Terme numero " & Integer'Image (No_Terme) & "="
                           & Float'Image (Terme_Courant) ) ;
      -- Calcul du terme suivant
      Terme_Courant := Terme_Courant * Raison ;
   end loop ;
end Mission3 ;
```

Déclaration des variables ($\in$ definition)

Bonjour, ce programme affiche les premiers termes d'une suite geometrique

Quelle est la valeur du premier terme ? ***0.001***
Quelle est la valeur de la raison ? ***10***
Combien de termes voulez-vous ? ***3***

Terme numero 1 = 1.00000E-03
Terme numero 2 = 1.00000E-02
Terme numero 3 = 1.00000E-01

Notez ici les messages d'erreur que vous rencontrez fréquemment et la manière de les corriger.

« Foo **is not visible** »

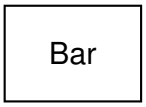
Structure d'un ***PROGRAMME*** *(p. 4)*

```
with Foo ;
procedure Mission is
   package F renames Foo ;
   -- Définitions
   D_i                                  ∈ definition
begin
   -- Corps du programme principal
   B ;                                  ∈ bloc
end Mission ;
```

PROCÉDURE *sans argument (p. 6)*

```
procedure Bar is
   -- Définitions éventuelles
begin
   -- Corps de la procédure
   B
end Bar ;
```

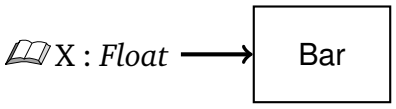
∈ definition



PROCÉDURE *avec arguments en entrée (p. 18)*

```
procedure Bar (X : Float) is
   -- Définitions éventuelles
begin
   -- Corps de la procédure
   B ;
end Bar ;
```

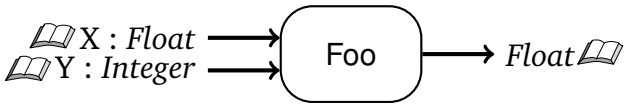
∈ definition



FONCTION *(p. 20), par exemple* $\mathbb{R} \times \mathbb{N} \longrightarrow \mathbb{R}$

```
function Foo (X : Float ; Y : Integer) return Float is
   -- Définitions éventuelles
   Resultat : Float ;
begin
   -- Corps de la fonction B ∈ bloc
   B ;
   -- Se termine forcément par return
   return Resultat ;
end Foo ;
```

∈ definition



Définitions de ***CONSTANTES*** *(p. 10)*

```
Pi               : constant Float   := 3.141593 ;
Vitesse_Lumiere  : constant Float   := 3.0E8 ;
Nombre_Eleves    : constant Integer := 462 ;
Nom_Appareil     : constant String  := "A380" ;
Est_En_Marche    : constant Boolean := False ;
```

∈ definition

VARIABLES *(p. 12)*

```
-- Définitions de variables ( ∈ definition )
Acceleration : Float ;
Compteur     : Integer ;
   ⋮
begin
   -- Affectations ( ∈ bloc )
   Acceleration := Force / Masse ;
   Acceleration := Lire_Accelerometre ;
   Acceleration := Acceleration * 1.05 ;
   -- Incrémente le compteur
   Compteur := Compteur + 1 ;
```

CONVERSION *(p. 14)*

Réel vers entier : Integer (e)
Entier vers réel : Float (e')

Bloc ***SÉQUENCE*** *(p. 8)*

$\mathcal{B}_1$;
$\mathcal{B}_2$;
⋮
$\mathcal{B}_n$;

∈ bloc

Bloc ***IF*** *(p. 24)*

```
if condition then
   B ;
else    -- ou elsif
   B' ;
end if ;
```

∈ bloc

Bloc ***FOR*** *(p. 26)*

```
for x in a .. b loop
   B(x) ;
end loop ;
```

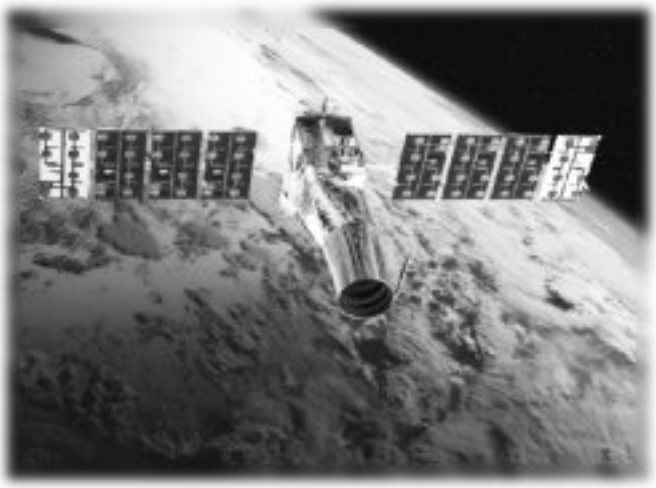
∈ bloc

Bloc ***WHILE*** *(p. 28)*

```
while condition loop
   B ;
end loop ;
```

∈ bloc

Quelques types : **Integer** (entiers) **Float** (réels) **Boolean** (vrai/faux) **String** (chaîne)



Ce document a été écrit avec emacs, compilé avec LaTeX, et utilise le package Ti*k*Z. Tous sont des logiciels libres.
Document compilé le 8 juillet 2021.

Fabriqué dans un atelier qui utilise : chocolat noir, chocolat noir aux noisettes, et biscuits divers.