Aide-mémoire Ada

UV Algorithmique 1
INSA première année

NOM :

....

PRÉNOM :

....

GROUPE :

....

Campagne 2008–2009

Didier LE BOTLAN
contact.lebotlan@insa-toulouse.fr
http://wwwdgeinew.insa-toulouse.fr/~lebotlan/

Introduction

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

STRUCTURER

- Structure d'un programme Ada — 4
- Procédures sans argument — 6
- Bloc Séquence — 8

CALCULER

- Les types de base — 8
- Les constantes — 10
- Les variables — 12
- Expression numérique — 14
- Conversion — 14
- Expression booléenne — 16

FONCTIONS ET PROCÉDURES AVEC ARGUMENTS

- Procédures avec argument — 18
- Fonctions — 20
- Types ARTICLE — 22

TESTER UNE CONDITION

- Bloc **IF** — 24

RÉPÉTER

- Bloc **FOR** — 26
- Bloc **WHILE** — 28

AFFICHER

- Les acteurs GAda.Text_IO — 30

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", "Jean_D_Ormesson", ou juste "X").

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 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.**

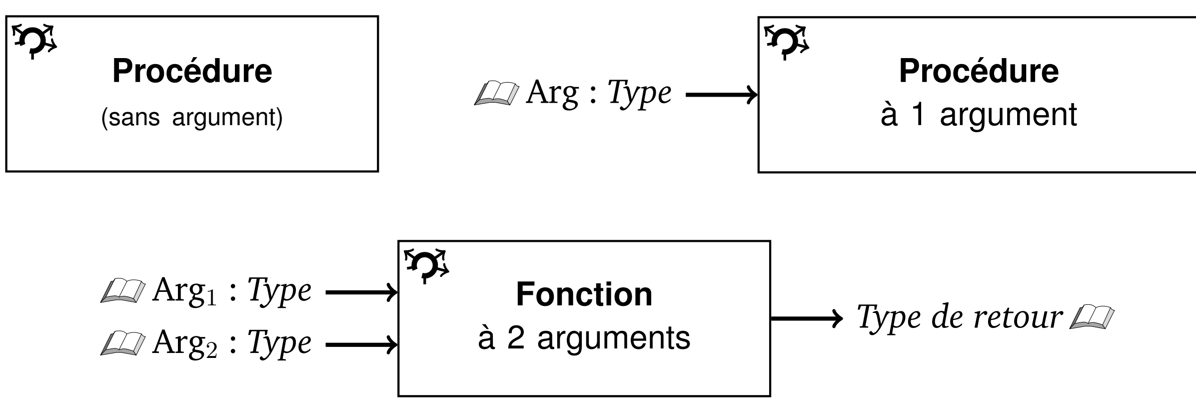
- **Un sous-programme** sert à factoriser un bloc de code en lui donnant un nom (p. ex. Foo). Le bloc pourra être réutilisé plusieurs fois en invoquant ce nom (appel du sous-programme Foo).

  Il existe deux types de sous-programmes : les **fonctions** et les **procédures**.

- **Le corps** d'un *programme* ou d'un *sous-programme* est le bloc de code définissant son comportement. Ce bloc est situé après le **begin**.

☞ 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.

☞ 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 deux formes :

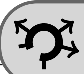
	⊢ e : *type*	pour indiquer le type d'une expression e
	⊢ terme : bloc	pour indiquer qu'un terme est un bloc de code.

Voici quelques exemples :

Jugement	*Interprétation*
⊢ 120 + 99 : *Integer*	120 + 99 est de type « *Integer* »
⊢ true : *Boolean*	true est de type « *Boolean* »
⊢ foo : *Integer*	La variable foo est de type « *Integer* »
⊢ foo > 42 : *Boolean*	foo > 42 est de type « *Boolean* »
⊢ foo := 42 ; : bloc	foo := 42 ; est un bloc de code

Pour alléger la notation, on écrit ⊢ foo := 42 : bloc (sans point-virgule).

Une expression n'est pas un bloc (et inversement) : ~~⊢ 120 + 99 : 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 des acteurs utilisés
with Acteur_1 , Acteur_2 ;
with ··· ;
procedure Mission is
    -- Renommage éventuel des acteurs
    package Bar renames Acteur_i ;
    ...
    -- Déclarations et définitions
    ...
begin
    -- Corps du programme principal
  ▷ 𝓑
end Mission ;
```

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

Exemple : le fichier mission1.adb

```
1  with GAda.Text_IO ;
3  procedure Mission1 is
5      package Txt renames GAda.Text_IO ;
7  begin
8      Txt.Put("Hello World !") ;
9  end Mission1 ;
```

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

ACTEUR "Foo"

Deux fichiers : foo.ads et foo.adb

Fichier foo.ads *(la spécification)*

```
-- Déclaration des acteurs utilisés
with Acteur_1 , Acteur_2 ;
with ··· ;
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 que sa signature, 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)*

```
package body Foo is
    -- Définitions des sous-programmes
    ...
end Foo ;
```

- Le corps de mission1.adb, ligne 8, est un bloc simple (appel de procédure).
- L'**indentation** est le décalage à droite de certaines lignes du programme (p. ex., lignes 5 et 8 de mission1.adb). Elle rend le code plus lisible en mettant en évidence les différentes parties (avant le **begin**, après le **begin**, l'imbrication des blocs).

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 courant.

Définition de procédure sans argument

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

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

On doit avoir $\vdash \mathcal{B} : \mathfrak{bloc}$

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("Bonjour, ") ;
10       Txt.Put("et bienvenue a l’INSA de ") ;
11    end Afficher_Bienvenue ;
13 begin
14    Afficher_Bienvenue ;
15    Txt.Put_Line("Toulouse") ;
17    Afficher_Bienvenue ;
18    Txt.Put_Line("Rennes") ;
20    Afficher_Bienvenue ;
21    Txt.Put_Line("Lyon") ;
22 end Mission2 ;
```

Définition de la procédure Afficher_Bienvenue

Afficher_Bienvenue

Invocations de la procédure

- ☞ Ce programme affiche trois messages de 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.

RÈGLE ⊢ « *Invocation* »

Une invocation de procédure est un bloc.

⊢ Moo.Bar : $\mathfrak{bloc}$

⊢ Bar : $\mathfrak{bloc}$

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 :

Définition du bloc séquence

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

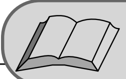
$$B_1\,;\quad B_2\,;\quad \vdots\quad B_n\,;$$

- ☞ 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_Diesel ;
procedure Mission1 is
begin
   Moteur_Diesel.Prechauffer ;
   Moteur_Diesel.Demarrer ;
   Moteur_Diesel.Laisser_Tourner ;
   Moteur_Diesel.Arreter ;
end Mission1 ;
```



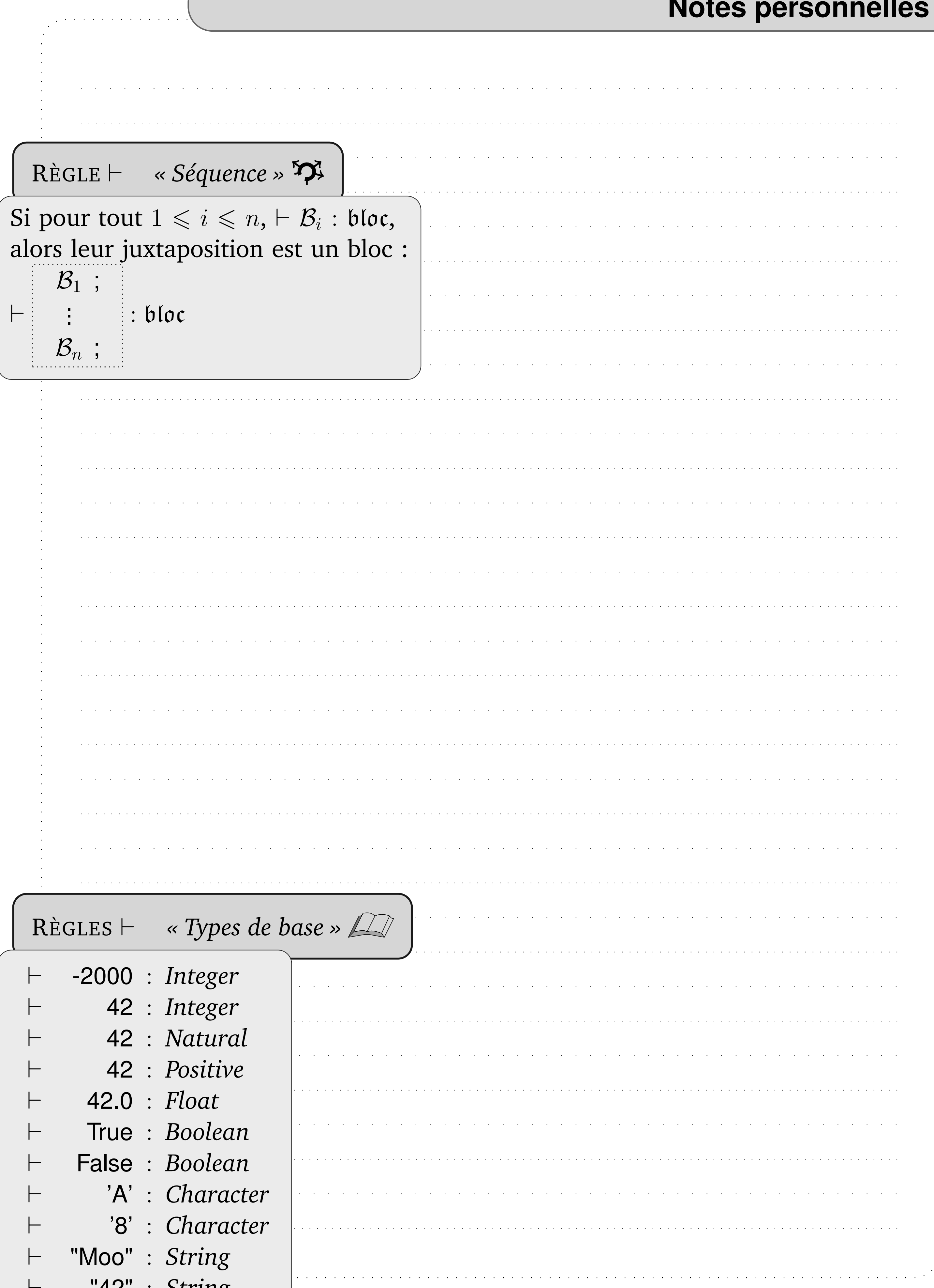
Les types de base

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."`	n octets (pour une chaîne de n caractères)

*Sur les machines 64 bits, un entier occupe 8 octets.

⅄Un réel occupe plus souvent 64 bits, y compris sur des machines 32 bits.



RÈGLE ⊢ *« Séquence »*

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

$$\vdash \begin{array}{l} \mathcal{B}_1 \ ; \\ \vdots \\ \mathcal{B}_n \ ; \end{array} : \mathfrak{bloc}$$

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 ;

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

Exemples de définitions de constantes

```
1  -- Ces définitions de constantes sont placées avant le begin du programme

3  Pi                         : constant Float    := 3.1415927 ;

5  -- Vitesse en m.s^-2
6  V_Lumiere                  : constant Float    := 3.0E8 ;

8  -- Vitesse en tours par minute
9  V_Rotation_Max             : constant Float    := 8200.0 ;

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

14 Nom_Appareil               : constant String   := "Airbus A380" ;

16 Est_En_Mode_Simulation     : constant Boolean  := False ;
```

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

(voir les lignes 3, 6, 8 de l'exemple).

RÈGLE ⊢ *« Constante »*

Après une définition de constante

Bar : **constant** *ce_type* : valeur

le jugement ⊢ Bar : *ce_type* est valide.

Par exemple :

⊢ Pi : *Float*
⊢ V_Lumiere : *Float*
⊢ V_Rotation_Max : *Float*
⊢ Nombre_Eleves : *Integer*
⊢ Nom_Appareil : *String*

Les variables

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

Variable de type *Integer*

· Pour DÉCLARER une variable Foo de type *Integer*, il suffit de placer la déclaration `Foo : Integer ;` avant le **begin** du programme.

· Pour changer la valeur de la variable, on utilise une AFFECTATION `Foo := expression numérique ;` par exemple `Foo := -25 + 160 ;`

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

Variables d'autres types

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

- Une **variable réelle** (de type *Float*) peut s'utiliser dans une expression numérique réelle (voir page 14).
- Une **variable booléenne** (de type *Boolean*) peut s'utiliser dans une expression booléenne (voir page 16).

✖ **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 ;
   -- Précision relative de la mesure de l'accélération (4%)
   Precision_Mesure : constant Float := 0.04

   Acceleration : Float ;                            } Déclaration de deux variables
   Acceleration_Est_Acceptable : Boolean ;
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

RÈGLE ⊢ « *Variable* »

Comme pour les constantes, cette déclaration implique ⊢ Foo : *Integer*

RÈGLE ⊢ « *Affectation* »

Si ⊢ Foo : *Integer* et ⊢ e : *Integer*
(c.-à-d., e est une expression numérique entière),

alors ⊢ Foo := e : 𝔟𝔩𝔬𝔠
(c.-à-d., une affectation est un **bloc**)

Expression numérique

Un calcul s'exprime à l'aide d'une **expression numérique**, c.-à-d. une formule, comme celle que l'on utiliserait sur une calculatrice :

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

☞ On peut UTILISER une expression numérique partout où le programme requiert un nombre, p. ex. dans une comparaison entre deux nombres :

```
if force * cos(30.0) / masse > 9.81 then ... end if ;
```

☞ 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).

Exemples d'expressions numériques

```
(Jour_Semaine + 1) mod 7
```

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

```
(35.0 * 35.0) / (50.5 - Vitesse)
```

Une expression numérique peut utiliser des constantes et des variables.

```
cos(30.0) * sin(Angle_Rotation) * sqrt(2.0)
```

sqrt est la fonction racine carrée

Les fonctions sin, cos, sqrt sont définies dans l'acteur Ada.Numerics.Generic_Elementary_Functions

```
2007 - Foo(1984) * Foo(1984 / 2)
```

Foo est une fonction définie auparavant dans le programme

(Voir la section sur les fonctions, page 20)

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.

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

Classification

$\vdash e : \textit{Integer}$	$\Rightarrow$	e expression numérique entière
$\vdash e : \textit{Float}$	$\Rightarrow$	e expression numérique réelle
$\vdash e : \textit{Boolean}$	$\Rightarrow$	e expression booléenne

RÈGLE $\vdash$ « *Opérateurs arithmétiques* »

Si e et e' sont deux expressions telles que $\vdash e : \textit{Integer}$ et $\vdash e' : \textit{Integer}$, alors

$\vdash e + e' : \textit{Integer}$
$\vdash e - e' : \textit{Integer}$
$\vdash e * e' : \textit{Integer}$
$\vdash e\,/\,e' : \textit{Integer}$ division entière
$\vdash e$ mod $e' : \textit{Integer}$
etc.

Idem pour les opérateurs sur les *Float*.

RÈGLE $\vdash$ « *Conversion* »

Si $\vdash e : \textit{Integer}$, alors $\vdash$ Float $(e) : \textit{Float}$
Si $\vdash e : \textit{Float}$, alors $\vdash$ Integer $(e) : \textit{Integer}$

RÈGLE $\vdash$ « *Puissance* »

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

Si $\vdash e : \textit{Float}$ et $\vdash e' : \textit{Integer}$, alors $\vdash e ** e' : \textit{Float}$.

Expression booléenne

Il est possible de calculer sur les booléens (valeurs Vrai et Faux).
Les opérateurs qui remplacent l'addition et la multiplication sont, respectivement, le **OU logique (OR)** et le **ET logique (AND)**. Par exemple :

```
Decollage_Possible : Boolean := Moteur_Vulcain_Pret and Satellite_OK ;
```

Moteur_Vulcain_Pret et Satellite_OK sont des variables ou des constantes de type *Boolean*

Les opérateurs de **comparaison** sur les nombres renvoient un booléen :

```
Acceleration_Est_Acceptable : Boolean := (Force / Masse) < (3.0 * 9.81) ;
```

Force et Masse sont des variables ou des constantes de type *Float*

Les expressions booléennes sont utiles dans un bloc **IF** (voir page 24).

Les opérateurs booléens à connaître

Opérateur	Signification	Sémantique	
OR	Ou logique	a **or** b	renvoie vrai si a **ou** b est vrai.
AND	Et logique	a **and** b	renvoie vrai si a **et** b sont vrais.
NOT	Non logique	**not** a	renvoie le contraire de a.

Les opérateurs de comparaison sur les nombres

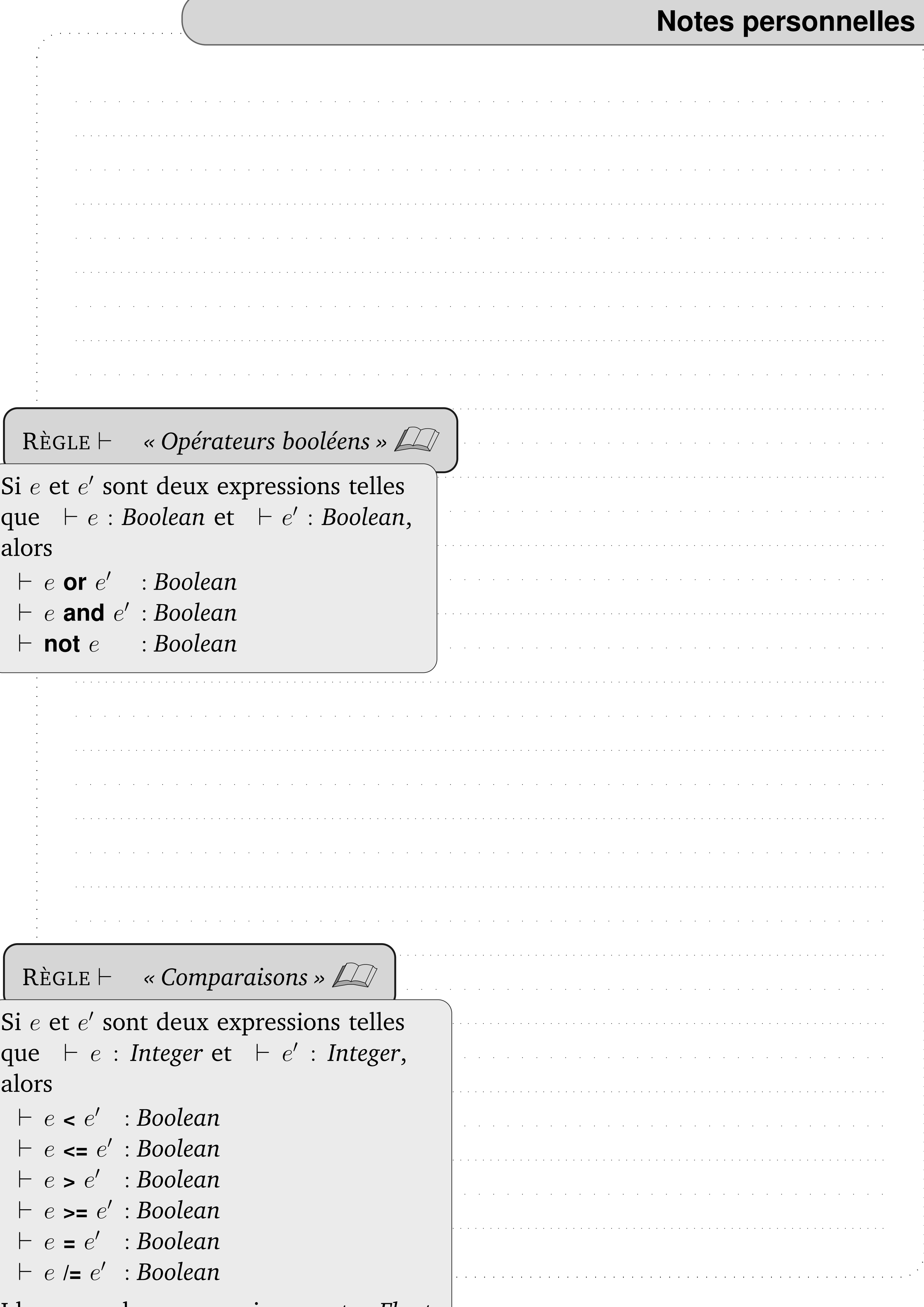
Opérateur	Signification
`<`	Inférieur strictement à
`<=`	Inférieur ou égal à
`>`	Supérieur strictement à
`>=`	Supérieur ou égal à
`=`	Égal à
`/=`	Différent de

RÈGLE : 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.



RÈGLE ⊢ « *Opérateurs booléens* »

Si e et e' sont deux expressions telles que $\vdash e$: *Boolean* et $\vdash e'$: *Boolean*, alors

$\vdash e$ **or** e' : *Boolean*
$\vdash e$ **and** e' : *Boolean*
$\vdash$ **not** e : *Boolean*

RÈGLE ⊢ « *Comparaisons* »

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

$\vdash e$ **<** e' : *Boolean*
$\vdash e$ **<=** e' : *Boolean*
$\vdash e$ **>** e' : *Boolean*
$\vdash e$ **>=** e' : *Boolean*
$\vdash e$ **=** e' : *Boolean*
$\vdash e$ **/=** e' : *Boolean*

Idem pour les comparaisons entre *Float*.

Procédures avec argument

Le corps d'une procédure peut dépendre de paramètres (que l'on appelle aussi « arguments ».

Définition de procédure à deux arguments (exemple)

```
procedure Foo (Bar₁ : Float ; Bar₂ : Integer) is
   -- Déclarations éventuelles
begin
   B (Bar₁, Bar₂)
end Foo ;
```

À placer avant le **begin**.

$\mathcal{B}$ doit être un bloc :
$\vdash \mathcal{B}\ (Bar_1, Bar_2)$: bloc

Bar₁ : *Float* → Foo
Bar₂ : *Integer* → Foo

Pour invoquer la procédure Foo, il faut fournir deux arguments (un réel et un entier), p. ex. `Foo (4.5, 120) ;`

RÈGLE ⊢ *« Appel de procédure »*

Un appel de procédure est un bloc :
Si $\vdash e : Float$ et $\vdash e' : Integer$ alors $\vdash$ Foo (e, e') : bloc

☞ Les trois invocations suivantes sont équivalentes :

```
Foo (4.5, 120) ;
Foo (Bar₁ => 4.5, Bar₂ => 120) ;
Foo (Bar₂ => 120, Bar₁ => 4.5) ;
```

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 ("Bonjour, ") ;
      Txt.Put_Line ("et bienvenue a l'INSA de " & Nom_INSA) ;
   end Afficher_Bienvenue ;
begin
   Afficher_Bienvenue ("Toulouse") ;
   Afficher_Bienvenue ("Rennes") ;
   Afficher_Bienvenue ("Lyon") ;
end Mission2 ;
```

`&` permet de coller deux chaînes

Une **FONCTION**, comme en mathématique, prend en entrée une ou plusieurs valeurs (que l'on appelle « arguments ») et renvoie une valeur en résultat.

$$Foo : \mathbb{R} \longrightarrow \mathbb{R}, \quad x \longmapsto x^3 + 1$$

x est l'argument, x^3+1 est le résultat

$$Bar : \mathbb{R}^2 \times \mathbb{N} \longrightarrow \mathbb{R}, \quad (x, y, n) \longmapsto x^n - y^n$$

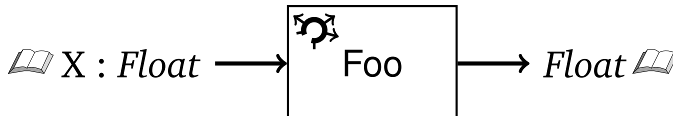
x, y, et n sont les arguments, $x^n - y^n$ est le résultat

Les fonctions sont utilisables dans les expressions numériques.
Par exemple ⊢ `100.0 + Foo (2.0) * Bar (11.0, -7.5, 3)` : *Float*.

Définition de fonction de $\mathbb{R}$ dans $\mathbb{R}$

```
function Foo (X : Float) return Float is
    -- Déclarations éventuelles
begin
    -- Corps de la fonction
    -- Se termine forcément par return
    return X**3 + 1 ;
end Foo ;
```

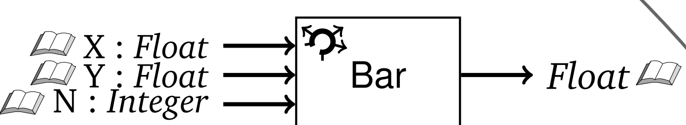
À placer avant le **begin**.



Une fonction à plusieurs arguments se définit en remplaçant la première ligne par :

Fonction à plusieurs arguments

```
function Bar (X, Y : Float ; N : Integer) return Float is
```



☞ La valeur retournée par une fonction n'est pas forcément un réel, cela pourrait être n'importe quel type, par exemple Integer ou Boolean.

RÈGLE *des* **Return**

Le premier **return** est suivi d'un type :
function Foo (...) **return** *ce_type* **is**

Le second **return** est suivi d'une expression de ce type :
return e ; avec ⊢ e : *ce_type*

RÈGLE ⊢ *« Appel de fonction »*

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

Si ⊢ e : *Float* alors ⊢ Foo (e) : *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
10    function Surface_Ellipse (Grand_Axe : Float ; Petit_Axe : Float) return Float is
11    begin
12       return (Pi * Grand_Axe * Petit_Axe) / 4.0 ;
13    end Surface_Ellipse ;

14    Grand_Axe : Float  --> Surface_Ellipse --> Float
      Petit_Axe : Float  -->

16
      -- DÉFINITION DE LA PROCÉDURE Affiche_Surface_Disque
17    procedure Affiche_Surface_Disque (Rayon : Float) is
18       Surface : Float ;
19    begin
20       Txt.Put ("La surface d'un disque de rayon " & Float'Image(Rayon)) ;
22       Surface := Surface_Ellipse (2.0*Rayon, 2.0*Rayon) ;   <-- Cette fonction est appelée ici
23       Txt.Put_Line (" est egale a " & Float'Image(Surface)) ;
24    end Affiche_Surface_Disque ;

25    Rayon : Float --> Affiche_Surface_Disque

26 begin
27    Affiche_Surface_Disque (1.0) ;   }
28    Affiche_Surface_Disque (2.0) ;   }  Trois appels de la procédure.
29    Affiche_Surface_Disque (3.0) ;   }
30 end Mission ;
```

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

```
La surface d'un disque de rayon 1.00000E+00 est egale a 3.14159E+00
La surface d'un disque de rayon 2.00000E+00 est egale a 1.25664E+01
La surface d'un disque de rayon 3.00000E+00 est egale a 2.82743E+01
```

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

- ⊢ Surface_Ellipse (10.0, 10.0) : *Float*
- ⊢ Affiche_Surface_Disque (5.0) : bloc

Types Article

Un **type article** est l'agrégation de plusieurs autres types.

Définition d'un type article

```
type Un_Foo is record
   Bar : un type ;
   ⋮
   Moo : un type ;
end record ;
```

(À placer avant le **begin**)

Accès aux attributs

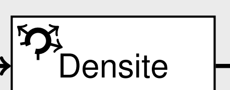
Si Zoo est une variable de type Un_Foo, l'expression `Zoo.Bar` renvoie la valeur de l'attribut Bar.

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

Exemple de type article

```
procedure Mission is
   type Des_Mesures is record
      Larg  : Float ;
      Long  : Float ;
      Masse : Integer ;
   end record ;
```

Définition du type Des_Mesures




```
   -- Fonction calculant la densité surfacique
   function Densite (Mes : Des_Mesures) return Float is
   begin
      return Float (Mes.Masse) / (Mes.Larg * Mes.Long) ;
   end Densite ;
```

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

```
   -- Déclaration d'une variable de type Des_Mesures
   Mesures_Plaque : Des_Mesures ;
begin
   -- Modification des champs, un par un
   Mesures_Plaque.Larg  := 12.0 ;
   Mesures_Plaque.Long  := 14.0 ;
   Mesures_Plaque.Masse := 6000 ;

   -- Modification de tous les champs
   Mesures_Plaque := (12.0, 14.0, 6000) ;

   -- Modification de tous les champs
   Mesures_Plaque := (Larg => 12.0, Long => 14.0, Masse => 6000) ;
end Mission ;
```

Ces trois manières de modifier les attributs sont équivalentes.

RÈGLE ⊢ « *Accès aux attributs* »

On suppose que le type article Un_Foo contient un attribut Bar de type *Integer*.

Si $\vdash e$: *Un_Foo* alors $\vdash e$. Bar : *Integer*

RÈGLE ⊢ « *Construction d'un article* »

On suppose que le type Des_Mesures est défini comme dans l'exemple ci-contre.

Si $\vdash e_1$: *Float*, $\vdash e_2$: *Float*, et $\vdash e_3$: *Integer*, alors

$\vdash (e_1, e_2, e_3)$: *Des_Mesures*

$\vdash$ (Larg => $e1$, Long => $e2$, Masse => $e3$) : *Des_Mesures*

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
   B
else
   B'
end if;
```

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

$\vdash \mathcal{B}$: bloc et $\vdash \mathcal{B}'$: 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 ;**

Exemples de blocs IF

```
if altitude > 45000.0 then Declencher_Alarme_Pilote ; end if ;
```

- ☞ altitude est une variable de type *Float*.
- ☞ L'alarme est déclenchée si, **au moment où est exécuté le bloc IF**, la variable altitude est supérieure à 45000.

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

Noter qu'il n'est pas possible d'écrire la condition ~~90 < Poids <= 100~~.

RÈGLE $\vdash$ « *Bloc* IF »

Un bloc **IF** est un bloc :

Si $\vdash e$: *Boolean*, $\vdash \mathcal{B}$: bloc et $\vdash \mathcal{B}'$: bloc,
alors $\vdash$ **if** e **then** $\mathcal{B}$ **else** $\mathcal{B}'$ **end if** : bloc

Notes personnelles

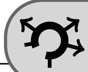
Dans un bloc **if**, le bloc $\mathcal{B}$ et le bloc $\mathcal{B}'$ sont des blocs quelconques. En particulier, $\mathcal{B}'$ peut être un bloc **if**, ce qui permet d'enchaîner les tests :

Exemple de blocs **IF** *imbriqués*

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

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 $[a..b]$.

(On dit que c'est un *itérateur*, car il itère sur l'intervalle.)

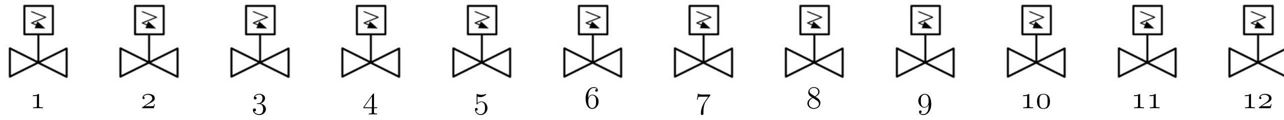
Définition du bloc FOR

Syntaxe :

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

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

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



Exemple de bloc FOR

```
-- Ferme les 12 électrovannes numérotées de 1 à 12.
for Numero_Electrovanne in 1..12 loop
   Fermer_Electrovanne (Numero_Electrovanne) ;
end loop ;
```

Équivalent au bloc séquence :

```
Fermer_Electrovanne (1) ;
Fermer_Electrovanne (2) ;
   ⋮
Fermer_Electrovanne (12) ;
```

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

```
-- Ce bloc for fait pivoter l'antenne de 58 degrés.
for N in 1..58 loop
   -- Fait pivoter l'antenne de 1 degré
   Pivoter_Antenne_1deg ;
end loop ;
```

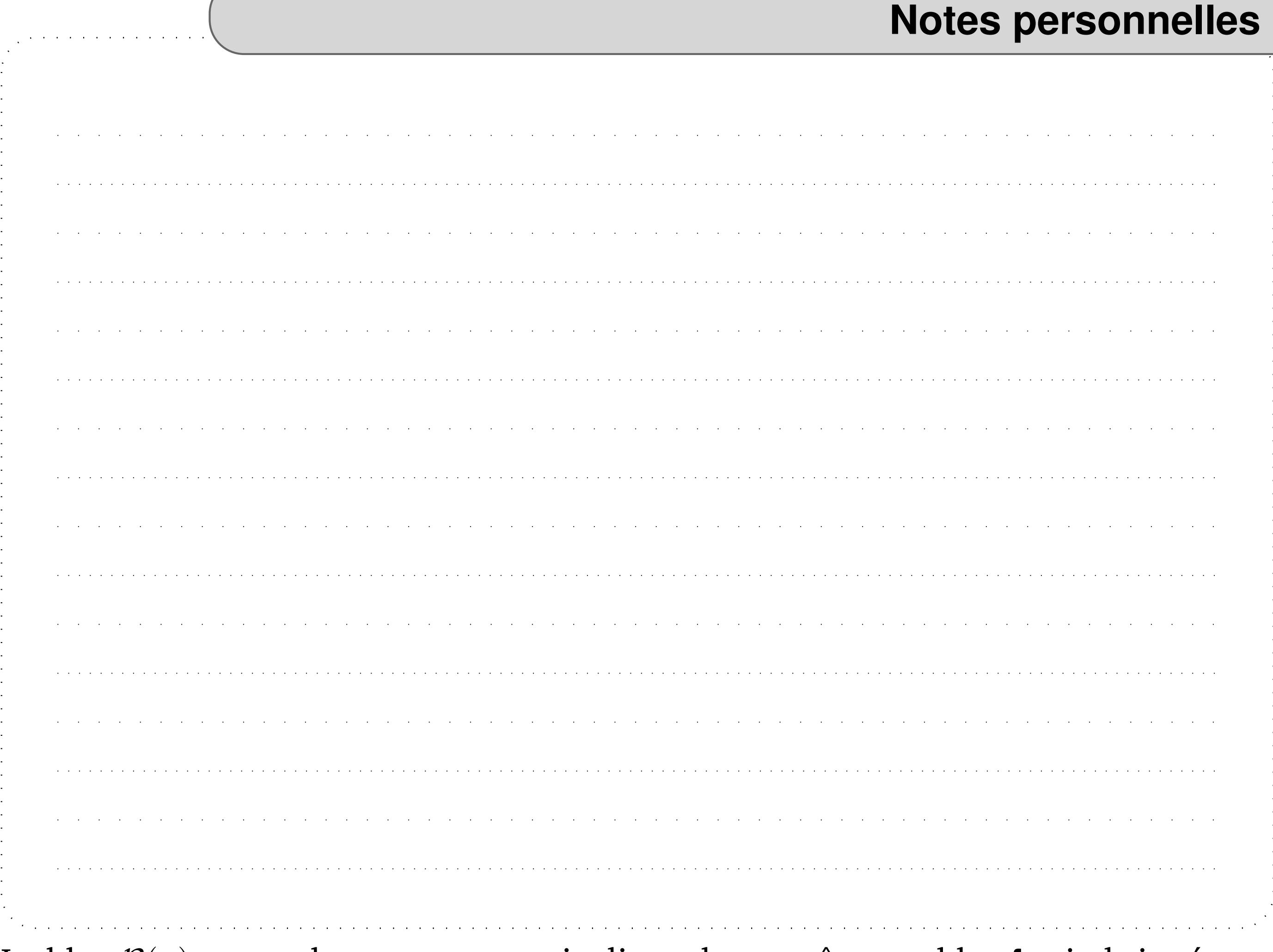
Noter que N n'est pas utilisé dans ce bloc **for**.

Règle $\vdash$ « *Bloc* FOR »

Un bloc **FOR** est un bloc :

Si $\vdash a$: *Integer*, $\vdash b$: *Integer* et $\vdash \mathcal{B}$: bloc,

alors $\vdash$ **for** x **in** a .. b **loop** $\mathcal{B}$ **end loop** : bloc



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 ;
begin
   for N_Ligne in 1..6 loop
      for N_Colonne in 1..N_Ligne loop
         Txt.Put( Integer'Image(N_Ligne) ) ;
         Txt.Put( Integer'Image(N_Colonne) ) ;
         Txt.Put( " --" ) ;
      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 --
```

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

Définition du bloc **WHILE**

Syntaxe :

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

avec $\vdash$ *condition* : *Boolean* (voir page 16)

et $\vdash \mathcal{B}$: 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.

Exemple de bloc **WHILE**

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

4  Niveau := Mesurer_Niveau_Cuve ; -- Mesure initiale

6  Ouvrir_Electrovanne ;

8  while Niveau < Capacite loop
9     Niveau := Mesurer_Niveau_Cuve ;
10 end loop ;

12 Fermer_Electrovanne ;
```

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 12) et le remplissage s'arrête.

☞ La mise à jour de la variable Niveau dans le bloc **while** (ligne 9) permet, au final, de modifier la valeur de la condition Niveau < Capacite, et de sortir du bloc.

☞ Cette mise à jour, ligne 9, est cruciale pour ne pas boucler indéfiniment (et ne pas faire déborder la cuve).

Notes personnelles

RÈGLE ⊢ « *Bloc* **WHILE** »

Un bloc **WHILE** est un bloc :

Si $\vdash e : Boolean$, et $\vdash \mathcal{B} : \mathfrak{bloc}$,

alors $\vdash$ **while** e **loop** $\mathcal{B}$ **end loop** $: \mathfrak{bloc}$

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 (Item : String) ;

-- Put_Line affiche le message et passe à la ligne
procedure Put_Line (Item : 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.
- 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 ( "La variable X vaut " & Integer'Image(X) )
```

Le programme sur la page ci-contre produit l'affichage suivant :

```
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 ? 4

Terme numero 1 = 1.00000E-03
Terme numero 2 = 1.00000E-02
Terme numero 3 = 1.00000E-01
Terme numero 4 = 1.00000E+00
```

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   ;  -- Premier terme de la suite
   Raison         : Float   ;  -- Raison de la suite géométrique
   Nombre_Termes  : Integer ;  -- Nombre de termes à afficher
   Terme_Courant  : Float   ;  -- Valeur du terme que l'on affiche
begin
   -- INTRODUCTION
   Txt.Put_Line ("Bonjour, ") ;
   Txt.Put_Line ("Ce programme affiche les premiers termes " &
                 "d'une suite geometrique") ;
   Txt.New_Line ;

   -- LECTURE DE LA VALEUR DU 1ER TERME
   Txt.Put ("Quelle est la valeur du premier terme ? ") ;
   Premier_Terme := GAda.Float_Text_IO.FGet ;

   -- LECTURE DE LA VALEUR DE LA RAISON
   Txt.Put ("Quelle est la valeur de la raison ? ") ;
   Raison := GAda.Float_Text_IO.FGet ;

   -- LECTURE DU NOMBRE DE TERMES
   Txt.Put ("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 ( "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

RÈGLE ⊢ *« Opérateurs et fonctions sur les chaînes »*

Si $\vdash e$: *String* et $\vdash e'$: *String*, alors $\vdash$ e & e' : *String*

Si $\vdash e$: *Integer*, alors $\vdash$ Integer'Image (e) : *String*

Si $\vdash e$: *Float*, alors $\vdash$ Float'Image (e) : *String*

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

« Foobar is not visible »

Vademecum

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

```
with Acteur₁ , Acteur₂ , ··· ;
procedure Foo is
   package Bar renames Acteurᵢ ;
   -- Définition(s) de constante
   -- Définition(s) de sous-programme
   -- Déclaration(s) de variable
begin
      -- Corps du programme principal
end Foo ;
```

***PROCÉDURE** sans argument (p. 6)*

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

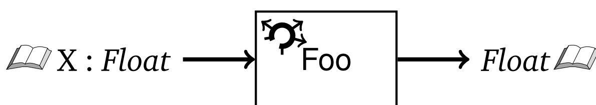
***PROCÉDURE** avec arguments (p. 20)*

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

X : *Float* → Bar

***FONCTION** de $\mathbb{R}$ dans $\mathbb{R}$ (p. 20)*

```
function Foo (X : Float) return Float is
   -- Déclarations éventuelles
begin
   -- Corps de la fonction
   -- Se termine forcément par return
   return X**3 + 1 ;
end Foo ;
```



*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 ;
```

***VARIABLES** (p. 12)*

```
-- Déclarations de variable (avant le begin)
Acceleration : Float ;
Compteur     : Integer ;
   ⋮
begin
   -- Affectations
   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 **IF** (p. 24)*

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

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

```
for x in 1 .. 12 loop
   B(x)
end loop ;
```

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

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

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 17 juillet 2008.

Fabriqué dans un atelier qui utilise : chocolat noir, chocolat noir aux amandes, aux noisettes, chocolat au lait, aux noisettes et raisin.