Mini Projet Informatique
IG2I L1 mars 2007

Structures de données

Algorithmique et programmation Ada

extrait du poly de cours 2000/2006 de SDA L1 de l'IG2I

Remerciements à Philippe Vanheeghe
Professeur à Centrale Lille

1 Découverte du Langage Ada et structures de base

1.1 Les identificateurs

Lors du développement d'un programme, il faut bien sûr donner des noms aux variables et constantes utilisées. Ces noms sont appelés identificateurs.

Pour donner un nom à une variable ou une constante (ou un type) en Ada, autrement dit pour écrire un identificateur en Ada, on écrit une suite de caractères qui commence obligatoirement par une lettre majuscule ou minuscule, et ensuite on peut faire figurer dans cette suite des lettres, majuscules ou minuscules, ou des chiffres. Dans le nom d'une variable en Ada, il est également possible d'utiliser le caractère trait bas (_).

Exemples:
`Compteur, index, La_Date` : sont des exemples d'identificateurs valides.
`Un+deux, _Toto` : sont des exemples d'identificateurs non valides.

1.2 Les littéraux numériques

Pour écrire un littéral numérique (un réel, un entier) en Ada, on écrit une suite de chiffres décimaux. Dans les littéraux réels, le caractère (.) est utilisé à la place du caractère (,). Dans l'écriture d'un littéral numérique on peut faire apparaître le caractère trait bas (_).

Exemples:
100, 1_000, 3.141_592_653_589 : sont des exemples de littéraux numériques valides.

1.3 La déclaration des variables

Avant de pouvoir utiliser une variable, il faut déclarer son nom (identificateur) et préciser son type, autrement dit il faut préciser l'ensemble dans lequel la variable va pouvoir prendre ses valeurs.

2 Les types standards du langage Ada sont :

INTEGER : une variable de type "INTEGER" prend ses valeurs dans IN,
FLOAT : une variable de type "FLOAT" prend ses valeurs dans IR,
LONG_FLOAT : une variable de type "LONG_FLOAT" prend ses valeurs dans IR,
BOOLEAN : une variable de type "BOOLEAN" prend soit la valeur vraie (TRUE en Ada) soit la valeur faux (FALSE en Ada),
CHARACTER : une variable de type "CHARACTER", prend sa valeur dans l'ensemble des caractères possibles. En Ada, un littéral caractère s'écrit, le caractère entre apostrophes, (par exemple 'A').
STRING(1..N) : il faut préciser le valeur de N, (la longueur maximum de la chaîne au moment de la déclaration), une variable de type "STRING(1..N)", prend sa valeur dans l'ensemble des chaînes de caractères de longueur N. En Ada un littéral chaîne de caractères s'écrit, une suite de N caractères au maximum entre guillemets, (par exemple "ceci est une chaîne de 36 caractères").

3 Exemples d'instructions de déclaration de variables en Ada :

`Compteur : INTEGER;`	déclare une variable, compteur, de type entier,
`Discriminant : FLOAT;`	déclare une variable, disciminant, comme réelle,
`Le_caractere : CHARACTER;`	déclare une variable, le_caractere, comme caractère.

Il faut remarquer le caractère (:) et le caractère (;) qui sont obligatoires dans ce type d'instruction.

Dans une instruction de déclaration il est possible d'initialiser la variable déclarée :

`Compteur : INTEGER` := 0; déclare une variable, compteur, de type entier, initialisée à zéro,
`Discriminant : FLOAT := 0.0`; déclare une variable, disciminant, comme réelle, initialisée à zéro,
`Le_caractere : CHARACTER := 'Y'` ; déclare une variable, le_caractere, comme caractère, initialisée avec le caractère Y.

Le langage Ada permet également la construction de types, c'est-à-dire de définir ses propres types. Les noms des types construits doivent respecter les règles de construction des identificateurs (cf. premier paragraphe).

3.1 La déclaration des constantes

La déclaration d'une constante en Ada est identique à la déclaration d'une variable, mais le mot réservé constant doit être mis avant le nom du type.

Exemples d'instructions de déclaration de constantes en Ada :

```
Un : constant integer := 1;
Pi : constant float := 3.14;
```

Il faut remarquer les caractères (:) et (;) qui sont obligatoires dans ce type d'instruction.

3.2 L'instruction d'affectation

La traduction de la primitive d'affectation, notée $\leftarrow$, se traduit en Ada par :=.

Exemples d'instructions d'affectation en Ada.

```
Nombre_D_Année := 10;
Valeur_Reelle := 12.3456;
Reponse := 'O';
```
Chaine_de_caractère := "Ceci est une chaîne de caractères";

Il faut remarquer que dans ces instructions le caractère (;) est obligatoire. Bien sûr avant de pouvoir écrire ces instructions, il faut avoir déclaré les variables, Nombre_D_Annee, Valeur_Reelle et Chaine_de_caractères. Une attention particulière doit être apportée à la déclaration de la variable Chaine_de_caractères; en effet, cette variable doit être déclarée comme une chaîne contenant exactement 33 caractères.

Le langage Ada étant un langage fortement typé, c'est-à-dire qu'un objet ne peut avoir qu'un et un seul type, une attention toute particulière doit être apportée lors de l'écriture des affectations. Par exemple l'affectation d'une valeur entière à une variable réelle est interdite, sauf si on réalise une conversion de type explicite.

3.3 Notion de programme Ada

Un programme Ada a la structure suivant:

Clauses de contexte
```
procedure mon_premier_programme_Ada is
-- Zone ou l'on déclare les variables, les constantes
begin
-- Traduction des actions de l'algorithme en Ada
end mon_premier_programme_Ada;
```

Les mots procedure, is, begin et end sont des mots réservé du langage; leur présence est obligatoire.
-- permet de commenter le programme.

Les clauses de contexte permettent de dire au compilateur quels sont les éléments dont il a besoin pour compiler le programme Ada. Par exemple, si dans le programme il est nécessaire de faire la lecture de la valeur d'une variable de type caractère (ou chaîne de caractères) au clavier, ou à l'inverse, l'impression de la valeur d'une variable de type caractère (ou chaîne de caractères) à l'écran, on trouvera dans les clauses de contexte, les instructions nécessaires à l'utilisation des opérations d'entrée-sortie sur les caractères (ou chaînes de caractères).

Les programmes doivent être clairs, faciles à relire.

3.4 Les mots réservés du langage Ada

Le liste des mots réservés du langage Ada est donnée dans le tableau suivant:

abort	delay	if	private	task
abs	delta	in	procedure	terminate
abstract	digits	is	protected	then
accept	do	limited	raise	type
access	else	loop	range	use
aliased	elsif	mod	record	until
all	end	new	rem	when
and	entry	not	renames	while
array	exception	null	requeue	with
at	exit	of	return	xor
begin	for	or	reverse	
body	function	others	select	
case	generic	out	separate	
constant	goto	package	subtype	
declare	if	pragma	tagged	

L'utilisation de ces mots comme identificateurs de variables est donc interdite.

3.4.1 Exempl

Clause de contexte permettant les entrées / sorties de caractères ou de chaînes de caractères.

```
  with Ada.Text_Io;
¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_1 is
 ªË¹¹¹¹¹¹¹¹
  §begin
  ¨¹¹ Ada.Text_Io. Put("Premier Programme Ada");
  ©end Exemple_1;
```

Un point-virgule est un terminateur d'instructions

Suite aux clauses de contexte, Put permet de faire l'impression à l'écran d'une chaîne de caractères, en utilisant le paquetage

3.4.2 Exemple 2

```
  with Ada.Text_Io;
¬¹¹¹¹¹¹¹¹¹
procedure Exemple_2 is
ªË¹¹¹¹¹¹¹¹
 § í Taille_Max :  constant Integer := 5;
 § í Chaine       :  String (1 .. Taille_Max);
 §begin
 ¨¹¹ Ada.Text_Io.Put("Entrez un chaine de caracteres ");
 ¨¹¹ Ada.Text_Io.Get(Chaine);
 ¨¹¹ Ada.Text_Io.Put("La chaine est: ");
 ¨¹¹ Ada.Text_Io.Put(Chaine);
 ©end Exemple_2;
```

Déclaration d'une constante, puis d'une variable

Suite aux clauses de contexte, Get permet de faire une saisie au clavier d'une chaîne de caractères.

3.4.3 Exemple 3

```
  with Ada.Text_Io;
¬¹¹¹¹¹¹¹¹¹
procedure Exemple_3 is
ªË¹¹¹¹¹¹¹¹
 § í Taille_Max : constant Integer := 5;
 § í Chaine     :          String (1 .. Taille_Max);
 §begin
 ¨¹¹ Ada.Text_Io.Put("Entrez une chaine de caracteres ");
 ¨¹¹ Ada.Text_Io.Get(Chaine);
 ¨¹¹ Ada.Text_Io.Put("La chaine est: ");
 ¨¹¹ Ada.Text_Io.New_line;
 ¨¹¹ Ada.Text_Io.Put(Chaine);
 ©end Exemple_3;
```

Suite aux clauses de contexte, New_line permet de faire un saut de ligne à l'écran

3.4.4 Exemple 4

```
  with Ada.Text_Io;
¬¹¹¹¹¹¹¹¹¹
procedure Exemple_4 is
ªË¹¹¹¹¹¹¹¹
 § ¡¥¥¥¥¥¥¥¥
 §ØÓìpackage Es_Entier is new Integer_Io(Integer);
 § £¦¦¦¦¦¦¦¦¦
 § í Valeur_Entiere : Integer := 0;
 §begin
 ¨¹¹ Ada.Text_Io.Put("La valeur initiale de l'entier est ");
 ¨¹¹ Es_Entier.Put(Valeur_Entiere);
 ¨¹¹ Ada.Text_Io.New_Line;
 ¨¹¹ Valeur_Entiere := Valeur_Entiere + 1;
 ¨¹¹ Ada.Text_Io. Put("La valeur de l'entier apres incrementation est ");
 ¨¹¹ Es_Entier.Put(Valeur_Entiere);
 ©end Exemple_4;
```

Suite aux clauses de contexte, cette instruction est obligatoire pour pouvoir utiliser les entrées-sorties sur les entiers.

Initialisation possible des variables à la déclaration

3.4.5 Exemple 5

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_5 is
 ªË¹¹¹¹¹¹¹¹
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Reel is new Float_Io(Float);
  § £¦¦¦¦¦¦¦¦¦
  § í Valeur_Reelle : Float := 0.0;
  §begin
  ¨¹¹ Ada.Text_Io.Put("La valeur initiale du reel est ");
  ¨¹¹ Es_Reel.Put(Valeur_Reelle);
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ Valeur_Reelle := Valeur_Reelle + 1.0;
  ¨¹¹ Ada.Text_Io.Put("La valeur du reel apres incrementation est ");
  ¨¹¹ Es_Reel.Put(Valeur_Reelle);
  ©end Exemple_5;
```

Suite aux clauses de contexte cette instruction est obligatoire pour pouvoir utiliser les entrées sorties sur les réels.

3.5 Les opérateurs de relation et les opérateurs logiques en Ada

Les opérateurs de relation en Ada sont:

= , opérateur égalité,
< , opérateur inférieur,
>, opérateur supérieur,
/=, opérateur différent,
<= , opérateur inférieur ou égal,
>= , opérateur supérieur ou égal.

Les résultats de ces opérations sont des booléens qui ont donc soit la valeur vraie (TRUE en Ada) soit la valeur faux (FALSE en Ada).

Les opérateurs logiques en Ada sont:

not, la négation,
or, le "ou" logique,
and, le "et" logique,
in, testant l'appartenance à un ensemble.

Les opérandes de ces 3 premiers opérateurs sont des booléens; le résultat est également un booléen.

3.6 Exemples d'opération logique en Ada.

A = B; donne comme résultat vrai ou faux selon que la valeur de la variable A est égale ou non à la valeur de la variable B.

(A = B) or (C = D); est une opération "ou" logique entre les opérandes (A = B) et (C = D). Le résultat d'une telle opération doit s'évaluer à partir des tables de vérité des opérateurs logiques.

3.6.1 Exemple 6

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_6 is
 ªË¹¹¹¹¹¹¹¹
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Booleen is new Enumeration_Io(Boolean);
  § £¦¦¦¦¦¦¦¦¦¦
  § í A, B, C, D : Integer;
  § í E, F, G : Boolean;
  §begin
  ¨¹¹ A:=1;
  ¨¹¹ B:=2;
  ¨¹¹ E:=A=B;
  ¨¹¹ Ada.Text_Io.Put("Le résultat du prédicat A = B est : ");
  ¨¹¹ Es_Booleen.Put(E);
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ C:=3;
  ¨¹¹ D:=4;
  ¨¹¹ F:=A<B and C>D;
  ¨¹¹ Ada.Text_Io.Put("Le résultat du prédicat A < B et C > D est : ");
  ¨¹¹ Es_Booleen.Put(F);
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ Ada.Text_Io.Put("Le résultat du prédicat non(A < B et C > D) est :
");
  ¨¹¹ Es_Booleen.Put(not(A<B and C>D));
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ G:= (A=B and C<=D-1) or A/=B;
  ¨¹¹ Ada.Text_Io.Put("Le résultat du prédicat (A=B and C<=D-1) or A/=B est
: ");
  ¨¹¹ Es_Booleen.Put(G);
  ©end Exemple_6;
```

Suite aux clauses de contexte cette instruction est obligatoire pour faire des entrées sorties de valeurs booléennes

3.7 Les opérateurs arithmétiques en Ada

Les opérateurs arithmétiques en Ada sont:

+, l'addition,
-, la soustraction,
-, le moins unaire, permettant de changer le signe d'une variable ou d'une constante,
/, la division,
*, la multiplication,
**, l'élévation à la puissance,
mod, le quotient de la division entière,
rem, le reste de le division entière.

Les parenthèses peuvent être utilisées dans l'écriture des expressions arithmétiques.

Le langage Ada étant un langage fortement typé, une attention toute particulière doit être apportée lors de l'écriture des expressions arithmétiques. Par exemple, la somme d'un entier et d'un réel est interdite (sans conversion explicite).

4 Exercices sur les éléments de base du langage Ada

4.1 Reprise des exemples précédents :

Reprendre les programmes précédents (exemple1 à exemple 5) et les faire fonctionner sur la machine. Une fois que vous aurez fait fonctionner un programme, vous le modifierez comme bon vous semble pour par exemple essayer d'autres opérateurs arithmétiques ou logiques. Mais attention : si vos modifications entraînent des erreurs, soit au moment de la compilation, soit au moment de l'exécution du programme, il convient de les expliquer soit par vous même soit avec l'aide de votre moniteur.

4.2 Autres exercices :

4.2.1 Exercice 1 :

Soit le programme suivant:

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exercice_1 is
 ªË¹¹¹¹¹¹¹¹
  §begin
  ¨¹¹ Ada.Text_Io. Put ("Utilisation du compilateur gnat");
  ¨¹¹ Ada.Text_Io. Put ("-------------------------------");
  ¨¹¹ Ada.Text_Io. Put ("1 - Vous devez sauvegardez votre programme");
  ¨¹¹ Ada.Text_Io. Put ("dans un fichier qui porte le meme mon");
  ¨¹¹ Ada.Text_Io. Put ("que votre programme (procedure) avec une extension
.adb");
  ¨¹¹ Ada.Text_Io. Put ("ici votre programme doit etre dans le fichier
Tp_1.adb");
  ¨¹¹ Ada.Text_Io. Put ("2 - Pour compiler votre programme cliquez sur
l'icone Compile" );
  ¨¹¹ Ada.Text_Io. Put ("3 - Pour construire un fichier executable
corresponant a votre programme" );   ¨¹¹ Ada.Text_Io. Put ("cliquez sur
l'icone Buid");
  ¨¹¹ Ada.Text_Io. Put ("4 - Pour executer votre programme cliquez sur
l'icone Run");
  ¨¹¹ Ada.Text_Io. Put ("L'icone Compile permet la compilation du
programme");
  ©end Exercice_1;
```

Essayer ce programme, puis le modifier pour que l'impression à l'écran soit mieux présentée.

4.2.2 Exercice 2 :

Soit le programme suivant:

```
  ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exercice_2 is
 ªË¹¹¹¹¹¹¹¹
  § í Entier,
  §   Resultat_Entier : Integer;
  § í Reel,
  §   Resultat_Reel   : Float;
  §begin
  ¨¹¹ Entier := 12;
  ¨¹¹ Reel := 10;
  ¨¹¹ Resultat_Entier := Entier + Entier;
  ¨¹¹ Resultat_Entier := Entier + Reel;
  ¨¹¹ Resultat_Reel := Reel + Reel;
```

```
  ¨¹¹ Resultat_Reel := Reel + Entier;
  ©end Exercice_2;
```

Essayer ce programme, puis expliquer les résultats obtenus lors de la compilation.

Quelles sont les instructions qu'il faut ajouter à ce programme pour pouvoir utiliser les instructions d'entrées-sorties sur les variables de type entier et les variables de type réel?

4.2.3 Exercice 3 :

Soit le programme suivant:

```
   with Ada.Text_Io;
 ¬¹11111111¹
Þßàprocedure Exercice_3 is
 ªË11111111
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Entier is new Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦¦
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Reel is new Float_Io(Float);
  § £¦¦¦¦¦¦¦¦¦
  § í Nombre_Entier,
  §   Resultat_Entier : Integer;
  § í Nombre_Reel,
  §   Resultat_Reel   : Float;
  §begin
  ¨¹¹ Nombre_Entier := 12;
  ¨¹¹ Nombre_Reel := 10.856;
  ¨¹¹ Resultat_Entier := Nombre_Entier + Nombre_Entier;
  ¨¹¹ Es_Entier.Put(Nombre_Entier);
  ¨¹¹ Ada.Text_Io.Put(" + ");
  ¨¹¹ Es_Entier.Put(Nombre_Entier);
  ¨¹¹ Ada.Text_Io.Put(" = ");
  ¨¹¹ Es_Entier.Put(Resultat_Entier);
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ Resultat_Entier := Nombre_Entier + Integer(Nombre_Reel);
  ¨¹¹ Ada.Text_Io.Put("La convertion du Reel : ");
  ¨¹¹ Es_Reel.Put(Nombre_Reel);
  ¨¹¹ Ada.Text_Io.Put(" en Entier donne : ");
  ¨¹¹ Es_Entier.Put(Integer(Nombre_Reel));
  ¨¹¹ Ada.Text_Io.New_line;
  ¨¹¹ Es_Entier.Put(Nombre_Entier);
  ¨¹¹ Ada.Text_Io.Put(" + ");
  ¨¹¹ Es_Reel.Put(Nombre_Reel);
  ¨¹¹ Ada.Text_Io.Put(" = ");
  ¨¹¹ Es_Entier.Put(Resultat_Entier);
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ Resultat_Reel := Nombre_Reel + Nombre_Reel;
  ¨¹¹ Es_Reel.Put(Nombre_Reel);
  ¨¹¹ Ada.Text_Io.Put(" + ");
  ¨¹¹ Es_Reel.Put(Nombre_Reel);
  ¨¹¹ Ada.Text_Io.Put(" = ");
  ¨¹¹ Es_Reel.Put(Resultat_Reel);
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ Resultat_Reel := Nombre_Reel + Float(Nombre_Entier);
  ¨¹¹ Ada.Text_Io.Put("La convertion de l'Entier : ");
  ¨¹¹ Es_Entier.Put(Nombre_Entier);
  ¨¹¹ Ada.Text_Io.Put(" en Reel donne : ");
  ¨¹¹ Es_Reel.Put(Float(Nombre_Entier));
  ¨¹¹ Ada.Text_Io.New_line;
```

```
  ¨¹¹ Es_Reel.Put(Nombre_Reel);
  ¨¹¹ Ada.Text_Io.Put(" + ");
  ¨¹¹ Es_Entier.Put(Nombre_Entier);
  ¨¹¹ Ada.Text_Io.Put(" = ");
  ¨¹¹ Es_Reel.Put(Resultat_Reel);
  ¨¹¹ Ada.Text_Io.New_Line;
  ©end Exercice_3;
```

Essayer ce programme, puis expliquer les résultats obtenus lors de la compilation.

4.2.4 Exercice 4 :

Soit le programme suivant:

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exercice_4 is
 ªË¹¹¹¹¹¹¹¹
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Entier is new Ada.Text_Io.Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦¦
  § í Mon_Entier : Integer;
  § í Ma_Chaine  : String(1..10);
  §begin
  ¨¹¹ Ada.Text_Io.Put("Entrer un entier : ");
  ¨¹¹ Es_Entier.Get(Mon_Entier);
  ¨¹¹ Ada.Text_Io.Put("Enter une chaine de caracteres : ");
  ¨¹¹ Ada.Text_Io.Get(Ma_Chaine);
  ¨¹¹ Ada.Text_Io.Put("Voici mon entier ");
  ¨¹¹ Es_Entier.Put(Mon_Entier);
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ Ada.Text_Io.Put("Voici ma chaine de caractere(s) ");
  ¨¹¹ Ada.Text_Io.Put(Ma_Chaine);
  ©end Exercice_4;
```

Essayer ce programme, puis expliquer les résultats obtenus lors de son exécution.

4.2.5 Exercice 5 :

Soit le programme suivant:

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exercice_5 is
 ªË¹¹¹¹¹¹¹¹
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Natural is new Ada.Text_Io.Integer_Io(Natural);
  § £¦¦¦¦¦¦¦¦¦
  § í Dernier   : Natural;
  § í Ma_Chaine : String (1 .. 20);
  §begin
  ¨¹¹ Ada.Text_Io.Put("Enter une chaine de caracteres : ");
  ¨¹¹ Ada.Text_Io.Get_Line(Ma_Chaine,Dernier);
  ¨¹¹ Ada.Text_Io.Put("Voici ma chaine de caractere(s) ");
  ¨¹¹ Ada.Text_Io.Put(Ma_Chaine);
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ Ada.Text_Io.Put("Voici la valeur de Dernier ");
  ¨¹¹ Es_Natural.Put(Dernier);
  ¨¹¹ Ada.Text_Io.New_Line;
```

```
  ¨¹¹ Ada.Text_Io.Put("Voici vraiment ma chaine de caractere(s) ");
  ¨¹¹ Ada.Text_Io.Put(Ma_Chaine(1..Dernier));
  ©end Exercice_5;
```

Essayer ce programme, puis expliquer les résultats obtenus lors de son exécution.

5 Exercice 4 :

Soit le programme suivant:

```
   with Ada.Text_Io;
 ¬¹11111111
Þßàprocedure Exercice_4 is
 ªË11111111
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Entier is new Ada.Text_Io.Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦¦
  § í Mon_Entier : Integer;
  § í Ma_Chaine  : String(1..10);
  §begin
  ¨¹¹ Ada.Text_Io.Put("Entrer un entier : ");
  ¨¹¹ Es_Entier.Get(Mon_Entier);
  ¨¹¹ Ada.Text_Io.Put("Enter une chaine de caracteres : ");
  ¨¹¹ Ada.Text_Io.Get(Ma_Chaine);
  ¨¹¹ Ada.Text_Io.Put("Voici mon entier ");
  ¨¹¹ Es_Entier.Put(Mon_Entier);
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ Ada.Text_Io.Put("Voici ma chaine de caractere(s) ");
  ¨¹¹ Ada.Text_Io.Put(Ma_Chaine);
  ©end Exercice_4;
```

Essayer ce programme, puis expliquer les résultats obtenus lors de son exécution.

6 Les structures de base de la programmation structurée en Ada

Ce paragraphe traite de la traduction des différentes structures de base de la programmation structurée par des instructions Ada.

6.1 La séquence

Une séquence est une suite d'actions qui seront exécutées les unes après les autres, ou encore séquentiellement.

Description de la séquence en pseudo langage :

Début
ACTION1
ACTION 2
.
.
ACTION n
Fin

Description de la séquence à l'aide du langage Ada :

Begin
Traduction de ACTION1;
Traduction de ACTION 2;
.
.
Traduction de ACTION n;
end;

Les n actions seront exécutées séquentiellement.

6.2 L'alternative

L'alternative permet de pouvoir exécuter une action ou une autre en fonction de la valeur d'un prédicat p.

Description de l'alternative en pseudo langage

si *p* ***alors***
ACTION ALORS
sinon
ACTION SINON
fsi

Description de l'alternative à l'aide du langage Ada

```
if Traduction du prédicat p then
        Traduction de ACTION ALORS;
else
        Traduction de ACTION SINON;
end if;
```

Lorsque l'on arrive dans une structure alternative, la première chose que l'on fait est l'évaluation du prédicat. Si l'évaluation du prédicat donne vrai on exécute l'action ALORS puis on sort de la structure. Si l'évaluation du prédicat donne faux, on n'exécute pas l'action ALORS, mais on exécute l'action SINON, puis on sort de la structure.

6.3 Le choix

Le choix permet de pouvoir exécuter une action ou non en fonction de la valeur d'un prédicat p.

Description du choix en pseudo langage

si *p* ***alors***
ACTION
fsi

Description du choix à l'aide du langage Ada

```
if Traduction du prédicat p then
        Traduction de ACTION;
end if;
```

Lorsque l'on arrive dans une structure choix, la première chose que l'on fait est l'évaluation du prédicat. Si l'évaluation du prédicat donne vrai on exécute l'action puis on sort de la structure. Si l'évaluation du prédicat donne faux, on sort de la structure.

6.4 Le choix multiple

Le choix multiple permet de pouvoir choisir d'exécuter une action parmi n en fonction de la valeur d'une variable s qui peut prendre ses valeurs dans l'ensemble $\{s_1, s_2, \ldots s_n\}$.

Description du choix multiple en pseudo langage

cas de *s* ***dans***
s_1 *: ACTION 1*
s_2 *: ACTION 2*
.
.
s_n *: ACTION n*
autre cas *: EXCEPTION*
fcas

Description du choix multiple à l'aide du langage Ada :

```
case Traduction de s is
 when s1 => Traduction de ACTION 1;
 when s2 => Traduction de ACTION 2;
                .
                .
```

when s_n => *Traduction de ACTION* n**;**

when others=>*Traduction de EXCEPTION;*
end case;

Lorsque l'on arrive dans une structure choix multiple, la première chose que l'on fait est l'évaluation de la variable s, appelée sélecteur du cas. Si l'évaluation de s donne une valeur appartenant à l'ensemble $\{s_1, s_2, \ldots s_n\}$, alors l'action correspondante est exécutée, puis on sort de la structure. Si l'évaluation de s donne une valeur n'appartenant pas à l'ensemble $\{s_1, s_2, \ldots s_n\}$, alors l'action EXCEPTION est exécutée puis on sort de la structure.

6.5 L'itérative

L'itérative permet de pouvoir exécuter une action tant qu'un prédicat p est vrai.

Description de l'itérative en pseudo langage

tant que *p* ***faire***
ACTION
fin tant que

Description de l'itérative à l'aide du langage Ada

while *Traduction du prédicat p* **loop**
Traduction de ACTION;
end loop;

Lorsque l'on arrive dans une structure itérative, la première chose que l'on fait est, l'évaluation du prédicat. Si l'évaluation du prédicat donne faux on n'exécute pas l'ACTION, on sort de la structure. Si l'évaluation du prédicat donne vrai alors l'ACTION est exécutée autant de fois qu'il est nécessaire pour que l'évaluation du prédicat donne faux. Ceci veut donc dire que l'ACTION doit modifiée la valeur du prédicat pour assurer le passage de sa valeur à faux au bout d'un nombre finis d'itérations.

6.6 La répétitive

La répétitive permet de pouvoir exécuter une action jusqu'à ce qu'un prédicat p devienne vrai.

Description de la répétitive en pseudo langage

répéter
ACTION
jusque *p*
fin répéter

Description de la répétitive à l'aide du langage Ada

loop
Traduction de ACTION;
exit when *Traduction du prédicat p;*
end loop;

Lorsque l'on arrive dans une structure répétitive, la première chose que l'on fait est l'exécution de l'ACTION, puis on évalue le prédicat. Si l'évaluation du prédicat donne faux alors on exécute à nouveau l'ACTION qui est exécutée autant de fois qu'il est nécessaire pour que l'évaluation du prédicat donne vrai. Si l'évaluation du prédicat donne vrai alors on sort de la structure. Ceci veut donc dire que l'ACTION doit modifier la valeur du prédicat pour assurer le passage de sa valeur à vrai au bout d'un nombre fini d'itérations.

6.7 La boucle à compteur

La boucle à compteur permet de pouvoir exécuter une action n fois, n étant connu.

Description de la boucle à compteur en pseudo langage :

pour *compteur* ← *valeur d'initialisation* ***à*** n ***pas*** *valeur du pas* ***faire***
ACTION
fin pour

Description de la boucle à compteur à l'aide du langage Ada :

for *compteur* ***in*** *[valeur d'initialisation .. n]* ***loop***
Traduction de ACTION;
end loop;
si la variable compteur prend successivement chaque valeur de l'intervalle dans l'ordre croissant.

for *compteur* `in reverse` *[valeur d'initialisation .. n]* ***loop***
Traduction de ACTION;
end loop;
pour un ordre décroissant.

A chaque valeur de "compteur" dans l'intervalle défini, il correspond une exécution de l'ACTION. Attention les valeurs de la variable pas autorisées sont 1 et –1; par défaut le pas est égal à 1.

6.8 Exercices sur la traduction des structures de base de la programmation structurée en Ada

Le but de ces exercices est de vous familiariser avec la construction en Ada des structures de base de la programmation structurée.

6.8.1 Exercice 1 :

Soit le programme suivant:

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exercice_1 is
 ªË¹¹¹¹¹¹¹¹
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Entier is new Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦¦¦
  § í Valeur_1,
  §   Valeur_2,
  §   Max      : Integer;
  §begin
  ¨¹¹ Ada.Text_Io.Put("Entrer une première valeur entiere : ");
  ¨¹¹ Es_Entier.Get(Valeur_1);
  ¨¹¹ Ada.Text_Io.Put("Entrer une Deuxième valeur entiere : ");
  ¨¹¹ Es_Entier.Get(Valeur_2);
  ¨¹³´if Valeur_1 > Valeur_2 then
  § 6¾¹¹ Max := Valeur_1;
  § ¶´else
  § ¸¾¹¹ Max := Valeur_2;
  § È end if;
  ¨¹¹ Ada.Text_Io.Put("La valeur du maximum est ");
  ¨¹¹ Es_Entier.Put(Max);
  ©end Exercice_1;
```

Essayer ce programme et commenter les résultats obtenus.

Que se passe-t-il si vous entrez au clavier une constante réelle (12.345 par exemple) au lieu d'une constante entière?

Transformer le programme précédent pour la détermination du maximum de 3 valeurs.

6.8.2 Exercice 2 :

Soit le programme suivant.

```
   with Ada.Text_Io;
```

```
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exercice_2 is
 ªË¹¹¹¹¹¹¹¹
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Entier is new Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦¦
  § í Valeur_1,
  §   Valeur_2,
  §   Max      : Integer;
  §begin
  ¨¹¹ Ada.Text_Io.Put("Entrer une première valeur entiere ? ");
  ¨¹¹ Es_Entier.Get(Valeur_1);
  ¨¹¹ Ada.Text_Io.Put("Entrer une Deuxième valeur entiere ? ");
  ¨¹¹ Es_Entier.Get(Valeur_2);
  ¨¹¹´case Valeur_1 > Valeur_2 is
  §  ÷¹¹´when True =>
  §  6  ¾¹¹ Max := Valeur_1;
  §  ÷¹¹´when False =>
  §  6  ¾¹¹ Max := Valeur_2;
  §  ÷¹¹´when others =>
  §  6  ¾¹¹ null;
  §  ¶end case;
  ¨¹¹ Ada.Text_Io.Put("La valeur du maximum est ");
  ¨¹¹ Es_Entier.Put(Max);
  ©end Exercice_2;
```

Ici la clause **when others** n'est pas nécessaire (toutes les valeurs possibles ont été traitées), mais si on la fait apparaître il faut utiliser l'instruction **null**, pour dire que l'on ne fait rien dans la clause when others.

Essayer ce programme et commenter les résultats obtenus.

6.8.3 Exercice 3 :

Soit le programme suivant:

```
  with Ada.Text_Io;
¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exercice_3 is
ªË¹¹¹¹¹¹¹¹
 § í C : Character;
 §begin
 ¨¹¹ Ada.Text_Io.Put("Entrer un caractere : ");
 ¨¹¹ Ada.Text_Io.Get(C);
 ¨¹¹´case C is
 §  ÷¹¹´when 'a'..'z' =>
 §  6  ¨¹¹ Ada.Text_Io.Put("Minuscule");
 §  6  ¾¹¹ Ada.Text_Io.New_Line;
 §  ÷¹¹´when 'A'..'Z' =>
 §  6  ¨¹¹ Ada.Text_Io.Put("Majuscule");
 §  6  ¾¹¹ Ada.Text_Io.New_Line;
 §  ÷¹¹´when '0'..'9' =>
 §  6  ¨¹¹ Ada.Text_Io.Put("Chiffre");
 §  6  ¾¹¹ Ada.Text_Io.New_Line;
 §  ÷¹¹´when others =>
 §  6  ¨¹¹ Ada.Text_Io.Put("Autre caractere");
 §  6  ¾¹¹ Ada.Text_Io.New_Line;
 §  ¶end case;
 ©end Exercice_3;
```

Essayer ce programme et commenter les résultats obtenus.

6.8.4 Exercice 4 :

Soit le programme suivant:

```
  with Ada.Text_Io;
¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exercice_4 is
ªË¹¹¹¹¹¹¹¹
 § í C : Character;
 §begin
 ¨¹¹ Ada.Text_Io.Put("Entrer un caractere (# pour finir) ");
 ¨¹¹ Ada.Text_Io.Get(C);
 ¨¹¹±while C /= '#' loop
 §  7¹¹´case C is
 §  5  ÷¹¹´when 'a'..'z' =>
 §  5  6  ¨¹¹ Ada.Text_Io.Put("Minuscule");
 §  5  6  ¾¹¹ Ada.Text_Io.New_Line;
 §  5  ÷¹¹´when 'A'..'Z' =>
 §  5  6  ¨¹¹ Ada.Text_Io.Put("Majuscule");
 §  5  6  ¾¹¹ Ada.Text_Io.New_Line;
 §  5  ÷¹¹´when '0'..'9' =>
 §  5  6  ¨¹¹ Ada.Text_Io.Put("Chiffre");
 §  5  6  ¾¹¹ Ada.Text_Io.New_Line;
 §  5  ÷¹¹´when others =>
 §  5  6  ¨¹¹ Ada.Text_Io.Put("Autre caractere");
 §  5  6  ¾¹¹ Ada.Text_Io.New_Line;
 §  5  ¶end case;
 §  7¹¹ Ada.Text_Io.Put("Entrer un caractere (# pour finir) ? ");
 §  7¹¹ Ada.Text_Io.Get(C);
 §  °end loop;
 ©end Exercice_4;
```

Essayer ce programme et commenter les résultats obtenus.

6.8.5 Exercice 5 :

Soit le programme suivant:

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exercice_5 is
 ªË¹¹¹¹¹¹¹¹
  § í C : Character;
  §begin
  ¨¹¹ Ada.Text_Io.Put("Entrer un caractere (# pour finir) ? ");
  ¨¹¹ Ada.Text_Io.Get(C);
  ¨¹¹®loop
  §  7¹¹´case C is
  §  5  ÷¹¹´when 'a'..'z' =>
  §  5  6  ¨¹¹ Ada.Text_Io.Put("Minuscule");
  §  5  6  ¾¹¹ Ada.Text_Io.New_Line;
  §  5  ÷¹¹´when 'A'..'Z' =>
  §  5  6  ¨¹¹ Ada.Text_Io.Put("Majuscule");
  §  5  6  ¾¹¹ Ada.Text_Io.New_Line;
  §  5  ÷¹¹´when '0'..'9' =>
  §  5  6  ¨¹¹ Ada.Text_Io.Put("Chiffre");
  §  5  6  ¾¹¹ Ada.Text_Io.New_Line;
  §  5  ÷¹¹´when others =>
  §  5  6  ¨¹¹ Ada.Text_Io.Put("Autre caractere");
  §  5  6  ¾¹¹ Ada.Text_Io.New_Line;
  §  5  ¶end case;
  §  7¹¹ Ada.Text_Io.Put("Entrer un caractere (# pour finir) ? ");
  §  7¹¹ Ada.Text_Io.Get(C);
  §Â¹Ç¹¹ exit when C = '#';
  §  °end loop;
  ©end Exercice_5;
```

Essayer ce programme et commenter les résultats obtenus. Que se passe-t-il en particulier si le premier caractère entré au clavier est #? Corriger ce problème.

6.8.6 Exercice 6 :

Soit le programme suivant:

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exercice_6 is
 ªË¹¹¹¹¹¹¹¹
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Entier is new Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦¦
  §begin
  ¨¹¹±for I in 1..10 loop
  §  7¹¹ Es_Entier.Put(I);
  §  °end loop;
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹±for I in reverse 10..1 loop
  §  7¹¹ Es_Entier.Put(I);
  §  °end loop;
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹±for I in reverse 1..10 loop
```

Attention: La variable de contrôle du pour n'a pas besoin d'être déclarée. Elle prend automatiquement le type des valeurs qui sont dans l'intervalle qui suit le in.

```
  §  7¹¹ Es_Entier.Put(I);
  §  °end loop;
  ¨¹¹ Ada.Text_Io.New_Line;
  ©end Exercice_6;
```

Essayer ce programme et commenter les résultats obtenus.

6.8.7 Exercice 7 :

Soit le programme suivant:

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exercice_7 is
 ªË¹¹¹¹¹¹¹¹
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Entier is new Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦¦
  § í Max : constant Integer := 10;
  § í I   :          Integer;
  §begin
  ¨¹¹ I:=1;
  ¨¹¹±while I<= Max loop
  §  7¹¹ Es_Entier.Put(I);
  §  7¹¹ I:=I+1;
  §  °end loop;
  ¨¹¹ Ada.Text_Io.New_Line;
  ©end Exercice_7;
```

Essayer ce programme et commenter les résultats obtenus.

6.8.8 Exercice 8 :

Soit le programme suivant:

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
ÞßàprocedureExercice_8 is
 ªË¹¹¹¹¹¹¹¹
  §-- Construction d'une boucle répéter
  §-- En utilisant les instructions loop et exit
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Entier is new Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦¦
  § í Max : constant Integer := 10;
  § í I   :          Integer;
  §begin
  ¨¹¹ I:=1;
  ¨¹¹®loop
  §  7¹¹ Es_Entier.Put(I);
  §  7¹¹ I:=I+1;
  §Â¹Ç¹¹ exit when I > Max;
  §  °end loop;
  ¨¹¹ Ada.Text_Io.New_Line;
  ©end Exercice_8;
```

-- est la marque de commentaire,-- doit apparaître en début de chaque ligne de commentaire, quelle que soit la place des

Essayer ce programme et commenter les résultats obtenus.

6.8.9 Exercice 9 : Bacchus !

Effectuer la programmation en Ada des 2 questions de l'exercice 1 du fascicule n°1 avec les conventions suivantes :

N'ayant pas encore étudié les types énumérés, nous représenterons les différents contenus possibles par des entiers : 0 pour vide, 1 pour vin, 2 pour eau. V1, V2 et V3 seront donc des variables entières.

Dans la question 2, on désire visualiser les différentes opérations effectuées (ex : verre 1 vidé dans verre 2).

6.8.10 Exercice 10 : Trace d'une matrice.

Le but de cet exercice est de calculer la trace d'une matrice dont les différents coefficients seront rentrés ligne par ligne par l'utilisateur (sans sauvegarde de la matrice). Une fois la matrice rentrée par l'utilisateur, le programme devra fournir la trace de celle-ci.

7 TP Types et Attributs

Ce poly de TP comporte une première partie de présentation qu'il est vivement conseillé de lire attentivement avant de passer aux parties comportant des exercices.

7.1 Introduction, définition des types et des attributs

Typer les objets que va utiliser un algorithme, c'est regrouper sous un même terme les objets semblables, c'est-à-dire ayant les mêmes propriétés, les mêmes plages de valeurs, et associés aux mêmes opérations.

Exemple : types standards déjà présentés dans le premier poly de TP : integer, float, long_float, boolean, string (1..N).

Boolean, par exemple, est un type Ada regroupant des variables ou constantes de valeurs "true" et "false", associées aux opérations not, and, or.

Toutes les variables et constantes en Ada possèdent une valeur, un nom et un type.

Retour sur un exemple du premier fascicule de TP :

Taille_Max : constant Integer := 5 ; → constante de valeur 5, de nom Taille_Max et de type Integer.

Chaine : string (1 .. Taille_Max) ; → variable de valeur définie dans le programme, de nom Chaine et de type chaine de caractères.

Le langage Ada est un langage fortement typé, ce qui veut dire qu'une variable, une constante ou un littéral appartiennent à un type et un seul (un entier n'est pas un réel par exemple). En typant le plus possible, on évite les erreurs conceptuelles décelées au niveau de la compilation par exemple.

Toute opération sur une variable ou une constante d'un type doit appartenir à l'ensemble des opérations associées au type concerné. Si on veut réaliser des opérations entre variables ou constantes de types différents, il faut réaliser une conversion explicite (c'est-à-dire l'indiquer dans le programme).

Exemple (conversion explicite, tirée d'un exemple du fascicule 1 de TP) :

```
   with Ada.Text_Io;
 ¬11111111
Þßàprocedure Conversion_Explicite is
 ªË11111111
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Entier is new Ada.Text_Io.Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Reel is new Ada.Text_Io.Float_Io(Float);
  § £¦¦¦¦¦¦¦¦
  § í Nombre_Entier,   Resultat_Entier : Integer;
  § í Nombre_Reel, Resultat_Reel    : Float;
```

```
§begin
¨¹¹ Nombre_Entier := 12;
¨¹¹ Nombre_Reel := 10.856;
§……
¨¹¹ Resultat_Entier := Nombre_Entier + Integer(Nombre_Reel);
¨¹¹ Resultat_Reel := Nombre_Reel + Float(Nombre_Entier);
§……
©end Conversion_Explicite;
```

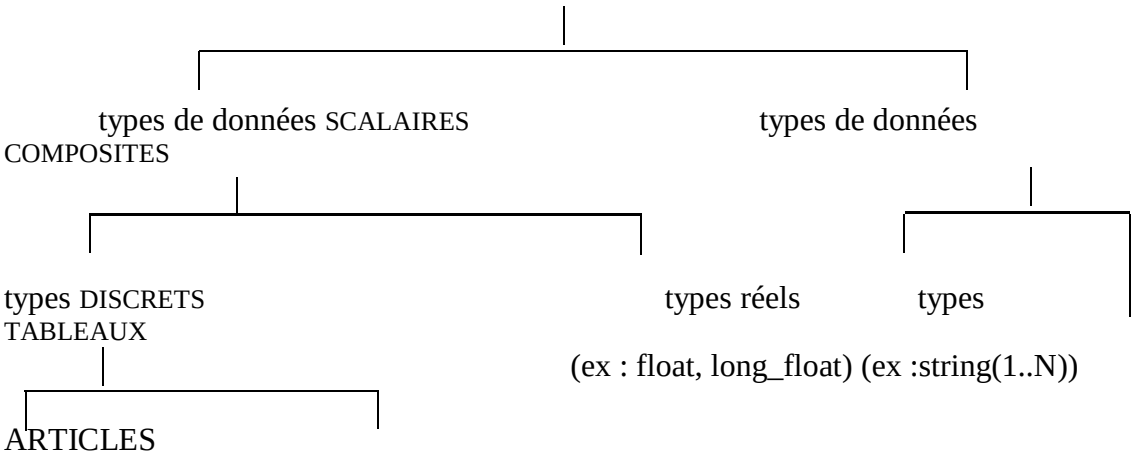
Ada (comme tout langage) propose des types prédéfinis (integer, float, …), mais permet également de construire ses propres types (on parlera alors de types construits), opération à privilégier le plus possible, afin d'améliorer la lisibilité du programme et d'éviter certaines erreurs.

Exemple : Les types standards présentés dans le premier fascicule de TP sont prédéfinis. Nous verrons dans la suite de ce TP comment construire de nouveaux types, par exemple un type jour comprenant les 7 jours de la semaine.

A chaque type est associé un ensemble d'attributs, accessibles par programmes. Les attributs décrivent certaines caractéristiques des types : valeurs minimale, maximale … On décrira par la suite les attributs des types standards.

7.2 Classification des principaux types :

Sont exclus de cette classification les quelques types non étudiés cette année, ainsi que les pointeurs.



Une variable de type scalaire ne présente qu'une valeur à un instant donné (toutes les variables présentées dans le premier fascicule de TP).

Une variable de type composite présente différentes valeurs à un même instant, valeurs qui sont toutes du même type dans le cas des tableaux, mais valeurs de types différents dans le cas des articles (voir paragraphe 5 et 6 de ce poly de TP).

En gras : types numériques.

7.3 Les types entiers et réels :

Les types entiers et réels prédéfinis sont bien sûr les types Integer et Float. On évitera toutefois de les utiliser au bénéfice de types construits car la précision de ces types prédéfinis dépend de la machine utilisée.

Le langage Ada permet de construire ses propres types entiers et réels, avec les contraintes souhaitées, notamment en matière de précision. Si la précision souhaitée n'est pas accessible sur la machine utilisée, le compilateur l'indiquera.

7.4 Les types entiers et réels construits :

Dans le cas des nombres entiers, on fournit un intervalle de travail :
Type Nom_du_type is range borne_inférieure .. borne_supérieure ;
Exemple :
type T_année is range 1950 ..2000 ;
type T_température is range –10 .. 40 ;
Rmq : Les deux types créés sont incompatibles entre eux.

On remarquera également à partir de ces exemples les autres avantages des types construits : moins d'espace mémoire "réquisitionné " car la plage de variation de la variable est précisée. De plus, si une opération non conforme est effectuée, conduisant la variable en dehors de sa plage de valeur, le programmeur ou l'utilisateur du programme sera prévenu.

Dans le cas des types réels :
Type point flottant : indication de l'erreur relative
type valeur_de_masse is digits 7 range 0.0 .. 3.0 ;
au moins 7 chiffres significatifs demandés.
Les réels suivants présentent 7 chiffres significatifs : 1234567=1.234567.106 ; 0.02000378=2.000378.10-2 ; 850600300= 8.506003.108.
Types point fixe : indication de l'erreur absolue
type voltage is delta 0.1 range –12.0 .. 12.0 ;
Si le résultat fourni est x, le résultat exact est compris entre x-0.1 et x+0.1.
Ces notions d'erreurs relative et absolue seront étudiées de manière plus approfondie au cours de l'année en analyse numérique.

7.4.1 Exercice 1 :

Ecrire un petit programme Ada faisant intervenir un type entier construit compris entre 100 et 200, un type réel compris entre 0.0 et 11.0 de précision absolue 0.2. Réaliser des opérations entre variables de ces deux types (pensez aux conversions explicites). Que se passe-t-il quand une opération conduit à un résultat à l'extérieur de la plage de valeurs indiquée dans la déclaration des types ?
Remarque : Pour les entrées-sorties de réels définis par une précision absolue, on utilisera le paquetage Ada.Text_Io.Fixed_Io au lieu de Ada.Text_Io.Float_Io.

7.4.2 Exercice 2 :

Donner un exemple de construction de type entier dont la précision n'est pas accessible sur votre machine.

7.5 Les opérations et attributs des types entiers et réels :

7.5.1 Propriétés, opérations et attributs communs aux types scalaires :

Opérations : affectation (:=), =, /=;
opérations liées à la relation d'ordre : <, >, <=, >=;
tests d'appartenance à un intervalle ou à un sous-type : in, not in;

Attributs : Les types scalaires possèdent une borne inférieure et une borne supérieure accessibles à l'aide des attributs FIRST et LAST.
Exemples : INTEGER'FIRST donne le plus petit entier stocké, INTEGER'LAST donne le plus grand entier stocké.
De manière générale, T'FIRST (resp. T'LAST) rend la borne inférieure (resp. supérieure) du type scalaire T ou du sous-type scalaire T (la notion de sous-type sera précisée ultérieurement).
Les valeurs de ces différents attributs dépendent de la machine utilisée dans le cas par exemple des types integer et float comme le montre l'exemple suivant :

7.5.2 Exercice 3 :

```
   with Ada.Text_Io ;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure exemple is
 ªË¹¹¹¹¹¹¹¹
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Entier is new Ada.Text_Io.Integer_Io (Integer) ;
  § £¦¦¦¦¦¦¦¦¦
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Reel is new Ada.Text_Io.Float_Io (Float) ;
  § £¦¦¦¦¦¦¦¦¦
  §begin
  ¨¹¹ Ada.Text_Io.Put ("  Valeur minimale des INTEGER : " ) ;
  ¨¹¹ Es_Entier.Put(INTEGER'FIRST) ;
  ¨¹¹ Ada.Text_Io.New_line ;
  §
  ¨¹¹ Ada.Text_Io.Put (" Valeur maximale des INTEGER : ") ;
  ¨¹¹ Es_Entier.Put(INTEGER'LAST) ;
  ¨¹¹ Ada.Text_Io.New_line ;
  §
  ¨¹¹ Ada.Text_Io.Put (" Valeur minimale des FLOAT : ") ;
  ¨¹¹ Es_Reel.Put(FLOAT'FIRST) ;
  ¨¹¹ Ada.Text_Io.New_line ;
  §
  ¨¹¹ Ada.Text_Io.Put (" Valeur maximale des FLOAT : ") ;
  ¨¹¹ Es_Reel.Put(FLOAT'LAST) ;
  ¨¹¹ Ada.Text_Io.New_line ;
  §
  ¨¹¹ Ada.Text_Io.Put (" Nombre de chiffres significatifs des FLOAT :
") ;
  ¨¹¹ Es_Entier.Put(FLOAT'DIGITS) ;
  ¨¹¹ Ada.Text_Io.New_line ;
  ©end exemple ;
```

Quelles sont les valeurs obtenues sur votre machine ?

Transformer le programme pour obtenir le même type d'informations pour des types entiers et réels construits.

Autres attributs :

SUCC, PRED : T'PRED(S) (respectivement T'SUCC(S)) donne le littéral qui est juste avant (respectivement après) celui contenu dans la variable S du type de base de T.

(toute valeur d'un type scalaire, sauf la dernière, possède un successeur. De même, toute valeur, sauf la première, possède un prédécesseur)

IMAGE et VALUE :

L'attribut Image permet de convertir un scalaire en la chaîne de caractère associée et Value permet de faire l'opération réciproque.

Exemple :

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_Image is
 ªË¹¹¹¹¹¹¹¹
  § í I : Integer;
  §begin
  ¨¹¹ I := 47;
  ¨¹¹ Ada.Text_Io.Put(Integer'Image(I));
```

```
©end Exemple_Image;
```

Integer'Image(I) est la chaine de caractère " 47 ". Cette instruction permet l'impression d'un entier à l'écran sans utilisation du paquetage Ada.Text_Io. Integer_Io (Integer) comme dans l'exemple 4 du premier poly de TP.

WIDTH : T'WIDTH rend la longueur maximale des images des valeurs du type ou du sous-type.
Par exemple, l'ajout dans le programme précédent de :
Es_Entier.Put (Integer'Width) ;
Conduit à l'affichage de : 11.

MAX, MIN :
T'MAX (S1,S2) rend la valeur maximale des valeurs de S1 et S2.
T'MIN (S1,S2) rend la valeur minimale des valeurs de S1 et S2.

SIZE : T étant un type (ou un sous-type) et S une instance du type T :
S'SIZE renvoie le nombre de bits d'implémentation de S (ne s'applique pas sur un type mais sur une instance du type).

BASE : Applicable à tous les types ou sous-types, pas seulement les types scalaires.

Exercice 4 : Ecrire un programme faisant intervenir tous les attributs présentés ici, sur des types entiers et/ou réels construits.

7.5.3 Propriétés et attributs spécifiques :

Les opérations également réalisables avec les types entiers et réels sont bien sûr les opérations mathématiques classiques (+, -, …).

Un attribut spécifique aux types réels :
FLOAT'DIGITS : donne le nombre de chiffres significatifs.

Attributs spécifics aux types discrets :
Toute valeur d'un type discret possède un rang => Attributs POS et VAL, qui seront détaillés dans le paragraphe concernant les types énumérés.

7.6 Les sous-types et les types dérivés :

7.6.1 Les sous-types :

Un sous-type caractérise un sous-ensemble, défini au moyen d'une contrainte, de valeurs d'un type, utilisant l'ensemble des opérations définies pour ce type (rmq : il est impossible de restreindre les opérations du type de base).
La contrainte s'exprime par le mot réservé range :
subtype Numero-de-jour is INTEGER range 1..31 ;
Noter la différence avec les types construits auparavant : présence du mot réservé subtype.
subtype T_majuscule is character range 'A' .. 'Z' ; *(Le type character est un type énuméré, cf. & 4).*

Rmq : La contrainte peut être introduite directement, sans passer par la définition d'un sous-type, dans la déclaration d'une variable.

Un sous-type n'étant pas un nouveau type, on peut affecter la valeur d'une variable du type de base à une variable d'un sous-type.
Exemple :

```
¬1111111111
Þßàprocedure Exemple_Sous_Type is
 ªË11111111
  § ï subtype Numero_de_jour is INTEGER range 1..31 ;
  § í A : Numero_de_jour ;
```

```
  § í B : Integer :=3 ;
  §begin
  ¨¹¹ A := B ;
  §-- OK
  ¨¹¹ B := 35 ;
  ¨¹¹ A := B ;
  §-- problème exécution
  ©end Exemple_Sous_Type;
```

Deux sous-types d'entiers sont déjà définis : les sous-types positive et natural.
subtype Positive is integer range 1 .. Integer'Last ;
subtype Natural is Integer range 0 .. Integer'Last ;

Remarque : la déclaration d'un sous-type ne définit pas un nouveau type.

7.7 Les types dérivés :

Exemple :
type Longueur is new FLOAT;
Le type Longueur est ici un type dérivé du type Float, qui est alors son type parent.
Caractéristiques d'un type dérivé :
Le type dérivé appartient à la même classe que le type parent.
L'ensemble des valeurs d'un type dérivé est une copie des valeurs du type parent. Une conversion explicite type dérivé ↔ type parent est possible.
Ses attributs sont les mêmes que ceux de son type parent.

7.8 Les types énumératifs :

Un type énumératif rassemble un ensemble fini d'éléments ordonnés.
Rappelons que les types boolean et character sont des types énumératifs.

7.8.1 Construction :

Syntaxe :
type Nom_du_type is (première_valeur, deuxième valeur, …, dernière valeur) ;

Exemples :
type T_Jour is (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche) ;
type T_couleur is (orange, rouge, jaune) ;
type fruit is (orange, pomme poire) ;

type couleur is (blanc,rouge,orange,jaune,vert,bleu,indigo,violet,noir);
subtype arc_en_ciel is couleur range rouge .. violet; -- sous-type énumératif

Remarque : ne pas confondre avec des chaînes de caractère.

Exemple d'utilisation :

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_Enum is
 ªË¹¹¹¹¹¹¹¹
  § ï type T_Jour is (lundi, mardi, mercredi, jeudi, vendredi, samedi,
dimanche) ;
  § í Aujourdhui, Demain : T_JOUR ;
  §
  §begin
  ¨¹¹ Aujourdhui := lundi ;
  ¨¹¹ Demain := T_Jour'Succ (Aujourd'hui) ;
```

```
  ¨¹¹ Ada.Text_Io.Put (T_Jour'Image (Demain)) ;
  ©end Exemple_Enum;
```

Que réalise ce programme ?

Types énumérés et surcharge :
Considérons les deux déclarations suivantes :
type couleur is (rouge, orange, vert, citron) ;
type fruit is (orange, pomme, poire, citron) ;
Il y a surcharge de ORANGE. Son type peut en général être déduit du contexte. Si ça n'est pas le cas, il faut par exemple le préciser par qualification de type : couleur'(orange) ; fruit'(orange).

Exemple :

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_Surcharge is
 ªË¹¹¹¹¹¹¹¹
  § ï type couleur is (rouge, orange, vert, citron) ;
  § ï type fruit is (orange, pomme, poire, citron) ;
  §begin
  ¨¹¹±for i in orange .. citron loop
  §  5--instructions
  §  °end loop;
  ©end Exemple_Surcharge;
```

Ceci conduit à une erreur de compilation : i : couleur ou fruit? Si i est une couleur, il prend successivement les valeurs orange, vert, citron. Si i est un fruit, il prend successivement les valeurs orange, pomme, poire, citron. Deux solutions à ce problème si i est une couleur:
Première solution :
for i in couleur range orange .. citron loop
……
end loop;
Deuxième solution :
for i in couleur'(orange) .. couleur'(citron) loop
……
end loop;

Un type énumératif est bien sûr ordonné. Les opérations de comparaison pourront donc être utilisées.

7.9 Opérations et attributs des types énumératifs :

7.9.1 Opérations et attributs communs à tous les types énumératifs :

Les types énumératifs étant des types scalaires, les opérations et attributs communs aux types scalaires sont valables dans le cas des types énumérés (voir paragraphe 2.2.1 de ce poly de TP).

Quelques exemples :
Si car est déclarée comme une variable de type Character, le test suivant peut être effectué :
if car in 'A' .. 'Z',
c'est-à-dire "si car est une majuscule".

T_JOUR'WIDTH vaut 8 à cause de mercredi, vendredi ou dimanche qui comportent 8 lettres.
T_couleur'pred (rouge) vaut orange.
T_Couleur'value ("rouge ") sera l'identificateur Rouge. T_Couleur'Image (rouge) sera la chaîne de caractère "rouge ". Utilité pour les entrées/sorties.

De même, les attributs communs aux types discrets sont bien sûr valables pour les types énumératifs :

T'POS (S) rend le "rang" de S (sa position, 0 à 255 pour les caractères) parmi les valeurs possibles (ordonnées) du type T, ou parmi celle du type de base du sous-type T. VAL est l'attribut réciproque.

Attention le rang (POS) du premier élément d'un énumératif est égal à 0. Par contre, l'attribut POS sur un type (ou un sous-type) entier est l'opérateur " identifié ". Ainsi, Integer'Pos (entier) est égal à la valeur de la variable entier. Par exemple Integer'pos (-5) vaut –5.

Exemple : Character'pos ('A') est égal à 65, et Character'val (65) est égal à 'A'.

7.9.2 Opérations spécifiques aux booléens :

Opérations logiques : not, and, or, xor (not est prioritaire sur les trois autres), and then, or else.

Remarques générales sur les attributs : POS, VAL, SUCC et PRED, agissant sur un sous-type, agissent en fait sur le type de base. Mais FIRST et LAST par contre font bien référence au sous-type.

7.9.3 Exercice 5 :

Compléter sur cette feuille (sans taper le programme pour l'instant) le programme qui suit en fonction des informations données dans ce polycopié (compléter notamment les commentaires).

Taper ensuite le programme complété par vos soins, et vérifier alors à l'exécution la pertinence des commentaires que vous aurez complétés.

```
   with Ada.Text_Io;
 ¬111111111
Þßàprocedure Exercice_5_Types is
 ªË11111111
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Entier is new Ada.Text_Io.Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Reel is new Ada.Text_Io.Float_Io(Float);
  § £¦¦¦¦¦¦¦¦
  § ï type T_Couleur is
  §      (Blanc, Rouge, Orange, Jaune, Vert, Bleu, Indigo, Violet,
Noir);
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Enum is new Ada.Text_Io.Enumeration_Io(T_Couleur);
  § £¦¦¦¦¦¦¦¦
  § ï subtype T_Arc_En_Ciel is T_Couleur range Rouge..Violet;
  § ï type Mes_Reels is digits 3;
  § ï type Volt is new Float range - 12.0 .. 12.0;
  § ï type Point_Fixe is delta 0.1 range - 1.0 .. 1.0;
  § ï type T_Heure is mod 24;
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Mes_Reels is new Ada.Text_Io.Float_Io(Mes_Reels);
  § £¦¦¦¦¦¦¦¦
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Ptf is new Ada.Text_Io.Fixed_Io (Point_Fixe);
  § £¦¦¦¦¦¦¦¦
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Heure is new Ada.Text_Io.Modular_Io (T_Heure);
  § £¦¦¦¦¦¦¦¦
  § í Couleur          : T_Couleur;
  § í Spectre          : T_Arc_En_Ciel;
  § í Valeur_Entiere   : Integer;
  § í Valeur_Reelle    : Float;
  § í Valeur_Mes_Reels : Mes_Reels;
  § í Tension          : Volt;
  § í Ptf              : Point_Fixe;
  § í Heure            : T_Heure;
```

```
§begin
¨11 Spectre:=Rouge;
¨11 Couleur:=T_Arc_En_Ciel'Base'First;
¨11 Ada.Text_Io.Put(T_Couleur'Image(Couleur));
§--   la ligne précédente imprime : _____________________
¨11 Ada.Text_Io.New_Line;
¨11 Couleur := Spectre;
§-- La ligne suivante produit CONSTRAINT_ERROR car_____________________
§-- Couleur := T_Arc_En_Ciel'Pred(Couleur);
¨11 Couleur:=T_Couleur'Pred(Couleur);
¨11 Es_Enum.Put(Couleur);
§--   la ligne précédente imprime : _____________________
¨11 Ada.Text_Io.New_Line;
¨11 Couleur := T_Couleur'Succ(Spectre);
¨11 Es_Enum.Put(Couleur);
§--   la ligne précédente imprime : _____________________
¨11 Ada.Text_Io.New_Line;
¨11 Es_Entier.Put(T_Couleur'Pos(Couleur));
§--   la ligne précédente imprime : _____________________
¨11 Ada.Text_Io.New_Line;
¨11 Es_Entier.Put(T_Couleur'Pos(T_Couleur'First));
§--   la ligne précédente imprime : _____________________
¨11 Ada.Text_Io.New_Line;
¨11 Es_Entier.Put(T_Arc_En_Ciel'Pos(T_Arc_En_Ciel'First));
§--   la ligne précédente imprime : _____________________
¨11 Ada.Text_Io.New_Line;
¨11 Valeur_Entiere:=4;
¨11 Es_Enum.Put(T_Couleur'Val(Valeur_Entiere));
§--   la ligne précédente imprime : _____________________
¨11 Ada.Text_Io.New_Line;
¨11 Es_Enum.Put(T_Arc_En_Ciel'Val(Valeur_Entiere));
§--   la ligne précédente imprime : _____________________
¨11 Ada.Text_Io.New_Line;
¨11 Es_Entier.Put(T_Couleur'Width);
§--   la ligne précédente imprime : _____________________
¨11 Ada.Text_Io.New_Line;   ¨11
Es_Enum.Put(T_Couleur'Min(Vert,Rouge));
§--   la ligne précédente imprime : _____________________
¨11 Ada.Text_Io.New_Line;
¨11 Es_Enum.Put(T_Couleur'Max(Vert,Rouge));
§--   la ligne précédente imprime : _____________________
¨11 Ada.Text_Io.New_Line;
¨11 Es_Entier.Put(Mes_Reels'digits);
§--   la ligne précédente imprime : _____________________
¨11 Ada.Text_Io.New_Line;
¨11 Valeur_Mes_Reels:=1.2345;
¨11 Es_Mes_Reels.Put(Valeur_Mes_Reels);
§--   la ligne précédente imprime : _____________________
¨11 Ada.Text_Io.New_Line;
¨11 Tension:=10.234;
¨11 Es_Reel.Put(Float(Tension));
§--   la ligne précédente imprime : _____________________
¨11 Ada.Text_Io.New_Line;
¨11 Es_Mes_Reels.Put(Mes_Reels(Tension));
§--   la ligne précédente imprime : _____________________
¨11 Ada.Text_Io.New_Line;
¨11 Valeur_Reelle := 1.2345;
¨11 Es_Entier.Put(Valeur_Reelle'Size);
§--   la ligne précédente imprime : _____________________
¨11 Ada.Text_Io.New_Line;
¨11 Ptf := 0.05;
¨11 Es_Ptf.Put(Ptf);
```

```
§--   la ligne précédente imprime : ____________________
¨11 Ada.Text_Io.New_Line;
¨11 Ptf := 0.5;
¨11 Es_Ptf.Put(Ptf);
§--   la ligne précédente imprime : ____________________
¨11 Ada.Text_Io.New_Line;
¨11 Ptf := Ptf + Point_Fixe'delta;
¨11 Es_Ptf.Put(Ptf);
§--   la ligne précédente imprime : ____________________
¨11 Ada.Text_Io.New_Line;
¨11 Ptf := Ptf + 0.16;
¨11 Es_Ptf.Put(Ptf);
§--   la ligne précédente imprime : ____________________
¨11 Ada.Text_Io.New_Line;
¨11 Ptf := 0.6;
¨11 Ptf := Ptf +0.14;
¨11 Es_Ptf.Put(Ptf);
§--   la ligne précédente imprime : ____________________
¨11 Ada.Text_Io.New_Line;
¨11 Ptf := 0.6;
¨11 Ptf := Ptf + 0.02;
¨11 Es_Ptf.Put(Ptf);
§--   la ligne précédente imprime : ____________________
¨11 Ada.Text_Io.New_Line;
¨11 Ptf := 0.6;
¨11 Ptf := Ptf + 0.08;
¨11 Es_Ptf.Put(Ptf);
§--   la ligne précédente imprime : ____________________
¨11 Ada.Text_Io.New_Line;
¨11 Es_Heure.Put(T_Heure'First);
§--   la ligne précédente imprime : ____________________
¨11 Ada.Text_Io.New_Line;
¨11 Es_Heure.Put(T_Heure'Last);
§--   la ligne précédente imprime : ____________________
¨11 Ada.Text_Io.New_Line;
¨11 Heure := T_Heure'Last;
¨11 Heure := Heure + 1;
¨11 Es_Heure.Put(Heure);
§--   la ligne précédente imprime : ____________________
¨11 Ada.Text_Io.New_Line;
©end Exercice_5_Types;
```

8 Les tableaux :

8.1 Déclaration :

Exemples : VECTEUR : array(1..10) of integer;

Cette instruction déclare une variable de type tableau, VECTEUR. C'est un tableau à une dimension, il faudra donc un seul indice ou index pour accéder à un élément ou composant du tableau; cet indice pourra prendre une valeur parmi 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, et aucune autre. Enfin, les éléments de ce tableau sont tous des entiers. Pour accéder à une valeur de la variable vecteur, il faut donner le nom du tableau, ici VECTEUR, et entre parenthèses la valeur de l'indice de l'élément que l'on veut manipuler. Cette valeur de l'indice peut être contenue dans une variable. Ainsi, `VECTEUR(5)`permet la manipulation de la valeur d'indice 5 dans le tableau, `VECTEUR(I)`, permet la manipulation de la valeur d'indice I dans le tableau.

Les indices d'un tableau (quelle que soit sa dimension) doivent appartenir à un type discret (énumératif ou entier).

Les types tableaux peuvent être déclarés de deux façons :
Tableaux contraints : Les domaines des indices sont spécifiés.
Par exemple :
Type Reel is digits 7 range –1.0 .. 3.2 ;
Type T_C_VECTEUR1 is array (1..10) of Reel;--T pour type, C pour contraint
Type T_C_VECTEUR2 is array (positive range 1..10) of Reel;
VECTEUR1 : T_C_VECTEUR2;

Tableaux non contraints : Les domaines des indices sont incomplètement spécifiés.
Par exemple :
Type T_NC_VECTEUR is array (Positive range <>) of Reel; (les indices sont des entiers strictement positifs)
<> indique que la précision de la plage de variation des indices est retardée.
La plage de variation de l'indice sera précisée à la déclaration de la variable ou lors de la déclaration d'un sous-type :
Soit :
VECTEUR2 : T_NC_VECTEUR (1..10);

Soit :
Subtype ST_INDICE is positive range 1..10;
Subtype ST_C_VECTEUR is T_NC_VECTEUR (ST_INDICE); --sous-type contraint
VECTEUR3 : ST_C_VECTEUR;

Les tableaux VECTEUR1, VECTEUR2 et VECTEUR3 sont tous trois des tableaux d'indices 1 à 10 et d'éléments réels. La déclaration de tableaux non contraints permet plus de souplesse, comme on le verra par la suite.

Les tableaux de dimensions supérieures sont déclarés de la même façon.
Exemple :
Type T_NC_MATRICE is array (POSITIVE range <>, POSITIVE range <>) of Reel;
MATRICE : T_NC_MATRICE (1..3,1..4);
--tableau 3 lignes, 4 colonnes

8.2 Exercices :

8.2.1 Exercice 6 :

Soit le programme suivant:

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exo_6 is
 ªË¹¹¹¹¹¹¹¹
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Es_Entier is new Ada.Text_Io.Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦¦
  § í Vecteur : array (1 .. 10) of Integer;
  §begin
  ¨¹¹±for I in 1..10 loop
  §  7¹¹ Ada.Text_Io.Put("Entrer la valeur numero ");
  §  7¹¹ Es_Entier.Put(I);
  §  7¹¹ Ada.Text_Io.Put(" du tableau vecteur.");
  §  7¹¹ Es_Entier.Get(Vecteur(I));
  §  °end loop;
  ¨¹¹±for I in 1..10 loop
  §  7¹¹ Ada.Text_Io.Put("La valeur numero ");
  §  7¹¹ Es_Entier.Put(I);
  §  7¹¹ Ada.Text_Io.Put(" du tableau vecteur est :");
  §  7¹¹ Es_Entier.Put(Vecteur(I));
  §  7¹¹ Ada.Text_Io.New_line;
```

```
§  °end loop;   ¨¹¹±for I in reverse 1..10 loop
§  7¹¹ Ada.Text_Io.Put("la valeur numero ");
§  7¹¹ Es_Entier.Put(I);
§  7¹¹ Ada.Text_Io.Put("du tableau vecteur est :");
§  7¹¹ Es_Entier.Put(Vecteur(I));
§  7¹¹ Ada.Text_Io.New_line;
§  °end loop;
©end Exo_6;
```

Essayer ce programme et commenter les résultats obtenus.

8.2.2 Exercice 7 :

Transformer ce programme en donnant un nom (Max_Element par exemple) à l'indice maximal du tableau. Quel est l'intérêt de cette modification ?

Transformer la déclaration de la variable vecteur de différentes manières, en utilisant par exemple des types non contraints.

8.2.3 Exercice 8 :

Remplacer dans une des deux dernières boucles Max_Element par Max_Element + 1, puis commentez les résultats obtenus.

8.2.4 Exercice 9 :

Ecrire un programme faisant intervenir deux matrices " Matrice " et " Resultat " d'indices entiers variant de 1 à Max_Element (constante entière), de composants réels (valeurs –1000 à 1000, 3 chiffres significatifs). Ce programme devra permettre :

*La saisie, **colonne par colonne**, des éléments du tableau Matrice. On désire que cette partie n'aie pas à être modifiée si la valeur de Max_Element est modifiée dans la déclaration. Les deux indices, de ligne et de colonne, seront précisés à l'utilisateur au fur et à mesure de la saisie.*

*L'affichage, **ligne par ligne**, des éléments de Matrice.*

Le calcul du carré de Matrice (cf. multiplication de deux matrices). La variable Résultat doit contenir le résultat de cette opération. Afficher le résultat ligne par ligne.

8.3 Opérations sur les tableaux :

Opérations d'affectation et tests d'égalité et d'inégalité : sur tous les tableaux de mêmes types de composants et d'indices, et de même dimension (pas forcément les mêmes valeurs d'indices).

Il est possible de ne considérer qu'une partie d'un tableau, par exemple de la manière suivante :

V1 := V3 (30 .. 40); *si V1 est bien sûr un tableau d'indices entiers au nombre de 11 (ne variant pas nécessairement de 30 à 40), de composants de même nature que ceux de V3.*

L'agrégation : C'est une opération spécifique aux types composites (tableaux et articles pour vous cette année). Elle est utilisée pour initialiser un élément ou pour modifier sa valeur dans le corps du programme. L'association d'un composant d'un tableau à une valeur peut se faire de 2 manières :

1- par position (dans l'ordre) :

```
type T_VECTEUR is array (INTEGER range <>) of INTEGER;
-- initialisation à la déclaration
a1 : T_VECTEUR (1..10) := (1,2,3,4,5,6,7,8,9,10);
a2 : T_VECTEUR (2..100) := (1,1,2,2,others=>0); --seuls les 4 premiers composants sont non nuls
a3 : T_VECTEUR (1..10);
…
begin
a3 :=(1, others =>0);
-- initialisation par affectation d'agrégat
```

-- équivalent à une boucle For parcourant chacun des indices du tableaux
…
2- par nomination : (on indique l'indice correspondant)
avec les déclarations précédentes :
a3 :=(5..10=>1,1 =>0, 2..4=>5);
-- tableau (0,5,5,5,1,1,1,1,1,1)
a3 :=(5..10=>1,1 =>0, others=>5);
-- comme pour l'initialisation par position, il est possible d'utiliser others, mais une seule fois, à la fin)

Exemple avec des tableaux bidimensionnels :
Type T_MATRICE is array(*POSITIVE* range <>, *POSITIVE* range <>) of integer;
MATRICE : T_MATRICE(1..2,1..3);
…
begin
…
-- ci-après, 4 initialisations equivalentes du tableau
MATRICE := (others=>(others=>2));
MATRICE := ((2,2),(2,2),(2,2));
MATRICE := ((2,2), (2,others=>2),(2,others=>2));
MATRICE := ((1..2=>2),(1..2=>2),(1..2=>2));

Les attributs des tableaux : surtout intéressant quand on travaille avec des tableaux non contraints.

T'FIRST et T'LAST donnent respectivement les valeurs des indices du premier et du dernier composants du tableau T,

T'RANGE donne l'intervalle de variation des indices du tableau (utile pour des boucles pour),

T'LENGTH donne le nombre d'éléments du tableau.

Dans le cas des tableaux de dimension supérieure à 1, on donnera entre parenthèses le rang de l'indice concerné. Ainsi, T'FIRST(2) sera la première valeur du deuxième indice du tableau.

Remarque : Le préfixe (T ici) n'est pas un type comme vu précédemment, mais l'identificateur du tableau.

8.4 Opérations spécifiques aux tableaux unidimensionnels :

<, >, <=, >= : si le type des composants est discret.

Un tableau vide sera toujours inférieur à un tableau qui ne l'est pas.

Dans le cas de tableaux A et B non vides, A < B dès qu'un composant de A est inférieur à son homologue de B, les composants étant parcourus par indices croissants. Il est ainsi possible de comparer des tableaux de tailles différentes.

Pour comprendre cette notion de comparaison, il suffit de se référer à l'ordre des mots dans un dictionnaire :"Algorithmique"<"Facile", de même "456">"2250".

Concaténation : de symbole &.

Exemples :
…
type T_VECTEUR is array (INTEGER range <>) of INTEGER;
a1 : T_VECTEUR (1..10) := (1,2,3,4,5,6,7,8,9,10);
a2 : T_VECTEUR (1..20) := (1,1,2,2,others=>0);
a3 : T_VECTEUR (1..30);
…
begin
a3 := a1&a2;
a1:= a3(1..5)&a3(25..29);
a2(1..5) := 5&a1(7..10);
…

8.4.1 Exercice 10 :

Ecrire un programme utilisant des tableaux unidimensionnels de composants entiers, d'indices entiers également, et permettant de visualiser toutes les opérations spécifiées dans les paragraphes 5.3 et 5.4.

8.5 Cas particuliers des chaînes de caractères :

Le type "String" est un type tableaux non contraints prédéfini à partir de la déclaration suivante (déjà réalisée dans Ada.Text_Io) :

Type STRING is array (POSITIVE range <>) of character;

Le type character étant un type énuméré, toutes les opérations précédemment présentées sont valables sur les chaînes de caractères.

8.5.1 Exercice 11 :

Soit le programme suivant :

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exo_11 is
 ªË¹¹¹¹¹¹¹¹
  § í Max_Element_Chaine_1 : constant Integer := 15;
  § í Max_Element_Chaine_2 : constant Integer := 15;
  § í Max_Element_Chaine_3 : constant Integer := Max_Element_Chaine_1
+ Max_Element_Chaine_2;
  § í Chaine_1           : String (1 .. Max_Element_Chaine_1);
  § í Chaine_2           : String (1 .. Max_Element_Chaine_2);
  § í Chaine_3, Chaine_4 : String (1 .. Max_Element_Chaine_3);
  § í Index              : Integer;
  §begin
  ¨¹¹ Chaine_3:="Le langage vert fut appele Ada";
  ¨¹¹ Ada.Text_Io.Put(Chaine_3);
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹±for I in 1 .. Max_Element_Chaine_3 loop
  §  7¹¹ Ada.Text_Io.Put(Chaine_3(I));
  §  °end loop;
  ¨¹¹ Ada.Text_Io.New_Line;
  §-- Affectation de tableaux
  ¨¹¹ Chaine_2:=Chaine_3(1..Max_Element_Chaine_2);
  ¨¹¹±for I in 1 .. Max_Element_Chaine_2 loop
  §  7¹¹ Ada.Text_Io.Put(Chaine_2(I));
  §  °end loop;
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ Index:=1;
  ¨¹¹±for I in 16..Max_Element_Chaine_3 loop
  §  7¹¹ Chaine_1(Index):=Chaine_3(I);
  §  7¹¹ Index:=Index+1;
  §  °end loop;
  ¨¹¹ Ada.Text_Io.Put(Chaine_1);
  ¨¹¹ Ada.Text_Io.New_Line;
  §-- Concaténation
  ¨¹¹ Chaine_4:=Chaine_2&Chaine_1;
  ¨¹¹±for I in 1 .. Max_Element_Chaine_3 loop
  §  7¹¹ Ada.Text_Io.Put(Chaine_4(I));
  §  °end loop;
  ©end Exo_11;
```

Essayer ce programme et commenter les résultats obtenus.

Réduire la taille de ce programme au maximum en remplaçant les boucle “ pour ” par des opérations globales.

8.6 Les articles (ou enregistrements) :

Un article (ou enregistrement) est un type composite qui permet d'associer des variables de types différents, contrairement aux tableaux qui associent des éléments (les composants) de même type. Les composants d'un article sont appelés " champs ".

Il existe des types articles contraints ou non. Nous n'étudierons ici que les types contraints ; les enregistrements à champs variants seront étudiés ultérieurement.

Exemple de déclaration :

```
Type T_PERSONNE is
Record
        Nom : string (1..30);
        Prénom : string (1..20);
        Age : natural;
End record;
```

Le type T_personne est un article comprenant trois champs : le champ nom qui est une chaîne de caractères, le champ prénom, et le champ âge qui est entier naturel.

Opérations pouvant être effectuées :
Affectations, tests d'égalité et d'inégalité.

Accès à un champ :
Suite à la déclaration de type précédente, il est possible de déclarer une variable PERSONNE de type T_PERSONNE, et d'accéder à chacun des champs en utilisant la notation pointée :
Par exemple *PERSONNE.Age := 20* permet d'affecter une valeur au troisième champ.

Les agrégats d'articles :
Permet de donner une valeur à chaque champ en une seule instruction. Les règles d'association sont identiques à celles des agrégats de tableaux (association par position, par nomination, ou mixte).
Rmq : L'association mixte n'était pas possible pour les tableaux.

Exemple :

```
type T_Complexe is
record
Re : Float := 0.0 ; -- partie reelle
Im : Float := 0.0 ; --partie imaginaire
end record ;
X : T_Complexe ;
```

On peut alors écrire :
X.Re := 30.1 ;
X.Im := 25.7 ;
ou de manière équivalente
X := (30.1,25.7) ;

8.6.1 Exercice 12:

Réaliser le programme dont les contraintes sont les suivantes :
Ce programme doit permettre de donner à l'utilisateur la date du lendemain du jour précisé (par l'utilisateur). S'il rentre lundi 5 décembre, le programme doit lui fournir à l'écran la date mardi 6 décembre. On supposera pour simplifier que tous les mois ont 30 jours.
Il sera créé 2 types énumérés pour les jours de la semaine et pour les mois de l'année, et un sous-type entier pouvant prendre les valeurs 1 à 30.
Un jour sera évidemment représenté par un article à 3 champs.
On utilisera au maximum les différents attributs présentés dans ce poly.

9 Les sous programmes

9.1 Les sous programmes un exemple introductif

Soit un ensemble E à n éléments. Supposons que l'on veuille calculer le nombre de parties à p éléments de E à l'aide de la formule: $C_n^p = \binom{n}{p} = \frac{n!}{(n-p)!\,p!}$. L'algorithme permettant de calculer cette grandeur peut être le suivant:

```
début
lire (n)
lire (p)
si n < 0 ou p < 0 ou p > n alors
écrire ("calcul impossible")
sinon
n_factorielle ← 1
pour i de 1 à n faire
n_factorielle ← n_factorielle * i
fin pour
n_moins_p ← n – p
n_moins_p_factorielle ← 1
pour i de 1 à n_moins_p faire
n_moins_p_factorielle ← n_moins_p_factorielle * i
fin pour
p_factorielle ← 1
pour i de 1 à p faire
p_factorielle ← p_factorielle * i
fin pour
c_n_p ← n_factorielle / n_moins_p_factorielle * p_factorielle
ecrire (c_n_p)
fin si
fin
```

Cet algorithme fait apparaître 3 fois le même sous – algorithme, celui du calcul de la factorielle.

Il serait intéressant de pouvoir définir une fois seulement cet algorithme du calcul de la factorielle, et ensuite de pouvoir l'utiliser pour calculer la factorielle de différentes valeurs. Comme la fonction factorielle sur une calculatrice. En algorithmique cette façon de faire conduit à la notion de **sous – programme**.

Il existe deux sortes de sous – programme

- les procédures,
- les fonctions.

Lorsque l'on parle de sous - programme en Ada il faut considérer deux choses:

- la définition de l'en tête du sous – programme, on dit son **profil**, ceci correspond à sa **spécification**,
- la définition proprement dite du sous - programme, c'est la partie algorithmique du sous – programme, qui défini son **corps**.

9.2 notions sur la définition de sous – programme

Le profil d'un sous – programme, représente ce qu'il faut connaître du sous – programme pour pouvoir l'utiliser dans un programme ou dans un autre sous – programme. Il faut définir son nom, et une liste de paramètres formels. La définition de cette liste de paramètres formels (qui peut être vide) doit faire apparaître le nom des paramètres, le type des paramètres et leur **mode de passage**.

Le mode de passage d'un paramètre consiste à préciser: si ce paramètre est vue comme une **entrée** par le sous – programme c'est le mode **in** en Ada, si ce paramètre est vue comme une **sortie** par le sous – programme c'est le mode **out** en Ada, si ce paramètre est vue comme une **entrée / sortie** par le sous – programme c'est le mode **in out** en Ada

Dans le cas d'une fonction en Ada, il faut en plus préciser le type des valeurs retournées par la fonction. Le profil d'une fonction Ada ne peut faire apparaître que des paramètres formels en mode **in**.

Si l'on considère l'exemple précédent et que l'on veut définir un sous programme pour le calcul de la factorielle il faut d'abord choisir si l'on veut définir une fonction ou une procédure.

9.2.1 une fonction qui calcule *n*!

Un profil pour cette fonction pourrait être le suivant:

```
 ¬111111111
ÛÜÝfunction Factorielle ( N : in Natural ) return Positive;
 a111111111
------------------------------------------------------------
-- Cette fonction calcule n!!, on suppose que
------------------------------------------------------------
```

Le mot réservé `function` annonce la définition d'une fonction, le nom de cette fonction est `factorielle`, la définition de cette fonction utilise un paramètre formel `N` qui est de type naturel et cette fonction retourne une valeur qui est un nombre entier positif.

Ce profil peut être considérer, comme une définition de la fonction, et doit être unique.

Ce type de définition, peut être utile lors de l'appel croisé de sous-programme, il ne faut pas oublier qu'en Ada, il faut qu'un objet soit défini avant de pouvoir l'utiliser.

Un corps pour cette fonction pourrait être le suivant:

```
 ¬111111111
ÞßàfunctionFactorielle (N : in Natural )return Positive is
 aË111111111
  §------------------------------------------------------------
  §-- Cette fonction calcule n!, on suppose que
  §------------------------------------------------------------
  § í Resultat : Positive := 1;
  §begin
  ¨11±for I in 1..N loop
  §  711 Resultat:=Resultat*I;
  §  °end loop;
Â1Ä11 return Resultat;
  ©end Factorielle;
```

Le corps de la fonction `factorielle`, représente le résultat de la traduction en langage Ada de l'algorithme du calcul de la factorielle.

Dans la définition du corps d'une fonction la partie du code Ada qui se trouve entre le mot réservé `is` et le premier `begin`, correspond à la **partie déclarative** du sous – programme. On trouvera dans cette partie déclarative la déclaration de tous les objets nécessaires à la définition du corps de la fonction.

Il faut remarquer que les objets déclarés dans cette partie déclarative, ne sont utilisable que dans le corps de la fonction, **on dit que ces objets sont locaux à la fonction**.

Dans notre exemple la variable positive `resultat` est locale à la fonction, et ne peux donc pas être utilisée ailleurs.

Pour utiliser une fonction il faut remplacer la liste des paramètres formels par la liste des paramètres effectifs, c'est à dire remplacer les paramètres formels, par les valeurs particulières sur lesquelles doit être effectuées le calcul. Bien sur cette substitution doit se faire en cohérence avec le profil de la fonction. Puis utiliser cette fonction dans une expression qui soit compatible avec le type de la valeur retournée par la fonction.

Un exemple d'utilisation de cette fonction pourrait être le suivant:

```
with Ada.Text_Io;
```

```
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_1 is
 ªË¹¹¹¹¹¹¹¹
  §
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Positif_Es is new Ada.Text_Io.Integer_Io(Positive);
  § £¦¦¦¦¦¦¦¦¦
  §
  § ¬¹¹¹¹¹¹¹¹¹
  §Þßàfunction Factorielle ( N : in Natural ) return Positive is
  § ªË¹¹¹¹¹¹¹¹
  §  §-----------------------------------------------------
  §  §-- Cette fonction calcule n!, on suppose que n est >= 0
  §  §-----------------------------------------------------
  §  § í Resultat : Positive := 1;
  §  §begin
  §  ¨¹¹±for I in 1..N loop
  §  §  7¹¹ Resultat:=Resultat*I;
  §  §  °end loop;
  §Â¹Ä¹¹ return Resultat;
  §  ©end Factorielle;
  §
  § í P : Natural := 3;
  §begin
  ¨¹¹ Positif_Es.Put(Factorielle(P));
  ©end Exemple_1;
```

Partie déclarative de la fonction

L'instruction return est nécessaire pour "retourner" le résultat du calcul par le nom de la fonction

On peut remarquer que dans cet exemple la définition de la spécification est omise, elle n'est pas obligatoire.

On peut remarquer qu'une fonction est une unité de compilation, c'est à dire que si l'on met le corps de la fonction `factorielle` dans un fichier fatorielle.adb, on peut compiler ce fichier seul.

A partir du moment ou la compilation de ce fichier est effectuée, la fonction `factorielle` doit être considérée comme faisant partie de la bibliothèque de sous – programmes Ada et peut donc être utilisée, comme le montre l'exemple suivant.

Le corps de la fonction `factorielle` est dans le fichier **factorielle.adb**.

```
 ¬¹¹¹¹¹¹¹¹¹
Þßàfunction Factorielle (N : in Natural )return Positive is
 ªË¹¹¹¹¹¹¹¹
  §-------------------------------------------------------------
  §-- Cette fonction calcule n!, on suppose que n est >= 0
  §-------------------------------------------------------------
  § í Resultat : Positive := 1;
  §begin
  ¨¹¹±for I in 1..N loop
  §  7¹¹ Resultat:=Resultat*I;
  §  °end loop;
Â¹Ä¹¹ return Resultat;
  ©end Factorielle;
```

Après compilation de ce fichier, l'utilisation de fonction `factorielle` est illustrée par l'exemple suivant:

```
   with Ada.Text_Io;
   with Factorielle;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_2 is
 ªË¹¹¹¹¹¹¹¹
```

```
  §
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Positif_Es is new Ada.Text_Io.Integer_Io(Positive);
  § £¦¦¦¦¦¦¦¦
  §
  § í P : Natural := 3;
  §begin
  ¨¹¹ Positif_Es.Put(Factorielle(P));
  ©end Exemple_2;
```

L'instruction `with Factorielle`, permet de dire au compilateur, que l'on va utiliser la fonction factorielle de la bibliothèque.

9.2.2 une procédure qui calcule *n*!

Un profil pour cette procédure pourrait être le suivant:

```
 ¬111111111
ÛÜÝprocedure Factorielle ( N : in Natural; R : out Positive );
 ª111111111
------------------------------------------------------------
-- Cette procédure calcule r=n!
------------------------------------------------------------
```

Le mot réservé `procédure` annonce la définition d'une procédure, le nom de cette fonction est `factorielle`, la définition de cette procédure utilise un paramètre formel en entrée `N` qui est de type naturel et formel en sortie `R` qui est un nombre entier positif.

Ce profil peut être considéré, comme une définition de la procédure, et doit être unique.

Ce type de définition, peut être utile lors de l'appel croisé de sous-programme, il ne faut pas oublier qu'en Ada, il faut qu'un objet soit défini avant de pouvoir l'utiliser.

Un corps pour cette procédure pourrait être le suivant:

```
¬111111111
ÞßàprocedureFactorielle ( N : in Natural; R : out Positive ) is
 ªË11111111
  §------------------------------------------------------------
  §-- Cette procédure calcule r=n!, on suppose
  §------------------------------------------------------------
  §begin
  ¨¹¹ R:=1;
  ¨¹¹±for I in 1..N loop
  §  7¹¹ R:=R*I;
  §  °end loop;
  ©end Factorielle;
```

Le corps de la procédure `factorielle`, représente le résultat de la traduction en langage Ada de l'algorithme du calcul de la factorielle.

Dans la définition du corps d'une procédure la partie du code Ada qui se trouve entre le mot réservé `is` et le premier `begin`, correspond à la **partie déclarative** du sous – programme. On trouvera dans cette partie déclarative la déclaration de tous les objets nécessaires à la définition du corps de la fonction. Ici cette partie déclarative est vide.

Il faut remarquer que les objets déclarés dans cette partie déclarative, ne sont utilisable que dans le corps de la procédure, **on dit que ces objets sont locaux à la procédure**.

Pour utiliser une procédure il faut remplacer la liste des paramètres formels par la liste des paramètres effectifs, c'est à dire remplacer les paramètres formels, par les valeurs particulières sur

lesquelles doit être effectuées le calcul. Bien sur cette substitution doit se faire en cohérence avec le profil de la procédure. Puis utiliser cette procédure comme une nouvelle primitive du langage.

Un exemple d'utilisation de cette procedure pourrait être le suivant:

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_3 is
 ªË¹¹¹¹¹¹¹¹
  §
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Positif_Es is new Ada.Text_Io.Integer_Io(Positive);
  § £¦¦¦¦¦¦¦¦¦
  §
  § ¬¹¹¹¹¹¹¹¹
  §Þßàprocedure Factorielle ( N : in Natural; R : out Positive ) is
  § ªË¹¹¹¹¹¹¹¹
  §  §-----------------------------------------------------
  §  §-- Cette procédure calcule r=n!, on suppose que
  §  §-----------------------------------------------------
  §  §begin
  §  ¨¹¹ R:=1;
  §  ¨¹¹±for I in 1..N loop
  §  §  7¹¹ R:=R*I;
  §  §  °end loop;
  §  ©end Factorielle;
  §
  § í P        : Natural := 3;
  § í Resultat : Positive;
  §begin
  ¨¹¹ Factorielle(P,Resultat);
  ¨¹¹ Positif_Es.Put(Resultat);
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ Factorielle(6,Resultat);
  ¨¹¹ Positif_Es.Put(Resultat);
  ©end Exemple_3;
```

On peut remarquer que dans cet exemple la définition de la spécification est omise, elle n'est pas obligatoire.

On peut remarquer qu'une procédure est une unité de compilation, c'est à dire que si l'on met le corps de la fonction `factorielle` dans un fichier fatorielle.adb, on peut compiler ce fichier seul.

A partir du moment ou la compilation de ce fichier est effectuée, la procedure `factorielle` doit être considérée comme faisant partie de la bibliothèque de sous – programmes Ada et peut donc être utilisée, comme le montre l'exemple suivant.

Le corps de la procedure `factorielle` est dans le fichier **factorielle.adb**.

```
¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Factorielle ( N : in Natural; R : out Positive ) is
 ªË¹¹¹¹¹¹¹¹
  §-------------------------------------------------------------
  §-- Cette procédure calcule r=n!, on suppose que
  §-------------------------------------------------------------
  §begin
  ¨¹¹ R:=1;
  ¨¹¹±for I in 1..N loop
  §  7¹¹ R:=R*I;
  §  °end loop;
  ©end Factorielle;
```

Après compilation de ce fichier, l'utilisation de la procédure `factorielle` est illustrée par l'exemple suivant:

```
   with Ada.Text_Io;
   with Factorielle;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_4 is
 ªË¹¹¹¹¹¹¹¹
  §
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Positif_Es is new Ada.Text_Io.Integer_Io(Positive);
  § £¦¦¦¦¦¦¦¦¦
  §
  § í P        : Natural  := 3;
  § í Resultat : Positive;
  §begin
  ¨¹¹ Factorielle(P,Resultat);
  ¨¹¹ Positif_Es.Put(Resultat);
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ Factorielle(6,Resultat);
  ¨¹¹ Positif_Es.Put(Resultat);
  ©end Exemple_4;
```

L'instruction `with Factorielle`, permet de dire au compilateur, que l'on va utiliser la procédure factorielle de la bibliothèque.

L'exemple suivant montre que le profil des sous – programmes est utilisé pour les différenciés.

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_5 is
 ªË¹¹¹¹¹¹¹¹
  §
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Positif_Es is new Ada.Text_Io.Integer_Io(Positive);
  § £¦¦¦¦¦¦¦¦¦
  §
  § ¬¹¹¹¹¹¹¹¹¹
  §Þßàfunction Factorielle ( N : in Natural ) return Positive is
  § ªË¹¹¹¹¹¹¹¹
  §  §----------------------------------------------------------
  §  §-- Cette fonction calcule n!, on suppose que
  §  §----------------------------------------------------------
  §  § í Resultat : Positive := 1;
  §  §begin
  §  ¨¹¹±for I in 1..N loop
  §  §  7¹¹ Resultat:=Resultat*I;
  §  §  °end loop;
  §Â¹Ä¹¹ return Resultat;
  §  ©end Factorielle;
  §
  § ¬¹¹¹¹¹¹¹¹¹
  §Þßàprocedure Factorielle ( N : in Natural; R : out Positive ) is
  § ªË¹¹¹¹¹¹¹¹
  §  §----------------------------------------------------------
  §  §-- Cette procédure calcule r=n!, on suppose que
  §  §----------------------------------------------------------
  §  §begin
  §  ¨¹¹ R:=1;
```

```
  §  ¨¹¹±for I in 1..N loop
  §  §  7¹¹ R:=R*I;
  §  §  °end loop;
  §  ©end Factorielle;
  §
  § í Resultat : Positive;
  §
  §begin
  ¨¹¹ Factorielle(4,Resultat);
  ¨¹¹ Positif_Es.Put(Resultat);
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ Resultat:=Factorielle(5);
  ¨¹¹ Positif_Es.Put(Resultat);
  ©end Exemple_5;
```

Dans cet exemple deux sous – programmes ont le même nom factorielle, ce sont les profils différents qui permet de différencier ces deux sous – programmes.

9.3 L'association des paramètres

Dans les exemples précédents, pour utiliser un sous – programme il faut remplacer la liste des paramètres formels par la liste des paramètres effectifs, c'est à dire remplacer les paramètres formels, par les valeurs particulières sur lesquelles doit être effectuées le calcul. Bien sur cette substitution doit se faire en cohérence avec le profil de la procédure, c'est à dire qu'il faut respecter la place des paramètres (on parle d'association par **position**).

En Ada, il est possible d'associer explicitement un paramètre effectif à son paramètre formel correspondant (on parle d'association **nominative**). A ce moment là le respect de l'ordre de définition des paramètres n'a plus à être respecté. l'exemple suivant illustre ce point particulier:

Il est possible dans l'appel d'un sous – programme d'utiliser ces 2 façons de faire pour associer un paramètre effectif à son paramètre formel correspondant. Mais attention on ne peut plus dans l'appel d'un sous – programme utiliser l'association par position, si on a commencer par des associations nominatives.

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_3 is
 ªË¹¹¹¹¹¹¹¹
  §
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Positif_Es is new Ada.Text_Io.Integer_Io(Positive);
  § £¦¦¦¦¦¦¦¦¦
  §
  § ¬¹¹¹¹¹¹¹¹¹
  §Þßàprocedure Factorielle ( N : in Natural; R : out Positive ) is
  § ªË¹¹¹¹¹¹¹¹
  §  §-------------------------------------------------------
  §  §-- Cette procédure calcule r=n!, on suppose que
  §  §-------------------------------------------------------
  §  §begin
  §  ¨¹¹ R:=1;
  §  ¨¹¹±for I in 1..N loop
  §  §  7¹¹ R:=R*I;
  §  §  °end loop;
  §  ©end Factorielle;
  §
  § í P        : Natural  := 3;
  § í Resultat : Positive;
  §begin
  ¨¹¹ Factorielle(
  §              N => P,
```

```
  §              R => Resultat);
  ¨¹¹ Positif_Es.Put(Resultat);
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ Factorielle(
  §              R => Resultat,
  §              N => 6);
  ¨¹¹ Positif_Es.Put(Resultat);
  ©end Exemple_3;
```

9.4 Les valeurs par défaut des paramètres

En Ada il est possible d'associer aux paramètres formels des valeurs par défaut. Ces valeurs sont attribuées aux paramètres formels lorsque les paramètres effectifs correspondant à ces paramètre formels sont omis lors de l'appel du sous - programme. Attention ceci n'est possible que pour des paramètres de mode **in**.

un exemple:

```
   with Ada.Text_Io;
 ¬¹11111111
Þßàprocedure Exemple_6 is
 ªË¹1111111
  §
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Entier_Es is new Ada.Text_Io.Integer_Io(Positive);
  § £¦¦¦¦¦¦¦¦
  §
  § ¬¹11111111
  §Þßàprocedure Ajouter (
  § §                  A : in     Integer;
  § §                  B :    out Integer;
  § §                  C : in     Positive := 1 ) is
  § ªË¹1111111
  §  §-------------------------------------------------------
  §  §-- Cette procédure calcule B=A+C
  §  §-------------------------------------------------------
  §  §begin
  §  ¨¹¹ B:=A+C;
  §  ©end Ajouter;
  §
  § í Valeur_Initiale : Integer  := 3;
  § í Resultat        : Integer;
  § í Valeur_Ajoutee  : Positive := 10;
  §begin
  ¨¹¹ Ajouter(Valeur_Initiale,Resultat);
  ¨¹¹ Entier_Es.Put(Resultat);
  ¨¹¹ Ajouter(Valeur_Initiale,Resultat,Valeur_Ajoutee);
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ Entier_Es.Put(Resultat);
  ¨¹¹ Ajouter(
  §          Valeur_Initiale,
  §          B => Resultat,
  §          C => Valeur_Ajoutee);
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ Entier_Es.Put(Resultat);
  ©end Exemple_6;
```

9.5 La surcharge

Le langage Ada, comme nous l'avons vu dans un exemple ci dessus, permet à plusieurs déclaration de posséder le même nom (identificateur). C'est ce que l'on appelle la surcharge. Quand un identificateur surchargé est utilisé, comme par exemple factorielle, qui représente à la fois une fonction et une procédure, le compilateur les différenties grâce à leurs profils, qui doivent être différents, sinon il signale l'ambiguïté.

Cette notion de surcharge est également valable pour les opérateurs suivants: +, -, /, *, mod, rem, **, abs, not, and, or, xor.

un exemple:

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_7 is
 ªË¹¹¹¹¹¹¹¹
  §
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Reel_Es is new Ada.Text_Io.Float_Io(Float);
  § £¦¦¦¦¦¦¦¦¦
  §
  § ï type T_Complexe is
  §      record
  §         Im : Float;
  §         Re : Float;
  §      end record;
  §
  § ¬¹¹¹¹¹¹¹¹¹
  §Þßàfunction "+" ( A, B : in T_Complexe ) return T_Complexe is
  § ªË¹¹¹¹¹¹¹¹
  §  §------------------------------------------------------------
  §  §-- Cette fonction calcule A+B, A et B étant des complexes
  §  §-- exprimés sous forme cartésienne
  §  §------------------------------------------------------------
  §  §begin
  §Â¹Ä¹¹ return (A.Re+B.Re,A.Im+B.Im);
  §  ©end "+";
  §
  § í C1 : T_Complexe := (1.0, 2.0);
  § í C2 : T_Complexe := (2.0, 3.0);
  § í C3 : T_Complexe;
  §
  §begin
  ¨¹¹ C3:=C1+C2;
  ¨¹¹ Reel_Es.Put(C3.Re);
  ¨¹¹ Reel_Es.Put(C3.Im);
  ©end Exemple_7;
```

9.6 La portée des déclarations, la visibilité des identificateurs

La portée d'un objet local au sous programme, c'est à dire déclaré dans la partie déclarative du sous – programme, en comprise entre la déclaration de l'objet et la fin du corps du sous – programme qui contient la déclaration de l'objet. Cet objet est donc inaccessible en dehors de cette zone, en particulier cet objet n'est pas accessible en dehors du sous – programme.

- La visibilité des identificateurs en Ada peut se résumer comme suit:
- un objet n'est pas visible tant qu'il n'est pas déclaré,

dans un sous programme les objets visibles sont ceux déclarés dans ce sous – programme, et les objets déclarés dans les corps des sous – programme qui l'englobent,

- la portée des objets déclarés dans un sous – programme est le sous – programme et à tous les sous – programmes déclarés dans ce sous – programme,
- soit un objet est déclaré dans un sous – programme. Si dans un autre sous – programme englobé dans le premier, un objet est déclaré avec ce même identificateur, alors cette définition masque la première.

Un exemple

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_8 is
 ªË¹¹¹¹¹¹¹¹
  §
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Entier_Es is new Ada.Text_Io.Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦¦
  §
  § í I : Integer := 1;
  § í L : Integer;
  § ¬¹¹¹¹¹¹¹¹¹
  §Þßàprocedure P1 is
  § ªË¹¹¹¹¹¹¹¹
  §  § í I : Integer := 2;
  §  § í J : Integer := 3;
  §  § ¬¹¹¹¹¹¹¹¹¹
  §  §Þßàprocedure P2 is
  §  § ªË¹¹¹¹¹¹¹¹
  §  §  § í J : Integer := 4;
  §  §  § í K : Integer;
  §  §  §begin
  §  §  ¨¹¹ Ada.Text_Io.Put_Line("*** Impression dans P2, apres begin
***");
  §  §  ¨¹¹ Entier_Es.Put(I);
  §  §  ¨¹¹ Ada.Text_Io.New_Line;
  §  §  ¨¹¹ Entier_Es.Put(J);
  §  §  ¨¹¹ Ada.Text_Io.New_Line;
  §  §  ¨¹¹ K:=5;
  §  §  ¨¹¹ J:=6;
  §  §  ¨¹¹ I:=7;
  §  §  ¨¹¹ L:=8;
  §  §  ¨¹¹ Ada.Text_Io.Put_Line("*** Impression dans P2, avant end
***");
  §  §  ¨¹¹ Entier_Es.Put(I);
  §  §  ¨¹¹ Ada.Text_Io.New_Line;
  §  §  ¨¹¹ Entier_Es.Put(J);
  §  §  ¨¹¹ Ada.Text_Io.New_Line;
  §  §  ¨¹¹ Entier_Es.Put(K);
  §  §  ¨¹¹ Ada.Text_Io.New_Line;
  §  §  ¨¹¹ Entier_Es.Put(L);
  §  §  ¨¹¹ Ada.Text_Io.New_Line;
  §  §  ©end P2;
  §  §begin
  §  ¨¹¹ Ada.Text_Io.Put_Line("*** Impression dans P1, avant P2 ***");
  §  ¨¹¹ Entier_Es.Put(I);
  §  ¨¹¹ Ada.Text_Io.New_Line;
  §  ¨¹¹ Entier_Es.Put(J);
  §  ¨¹¹ Ada.Text_Io.New_Line;
  §  ¨¹¹ P2;
  §  ¨¹¹ Ada.Text_Io.Put_Line("*** Impression dans P1, apres P2 ***");
  §  ¨¹¹ Entier_Es.Put(I);
  §  ¨¹¹ Ada.Text_Io.New_Line;
```

```
  §  ¨¹¹ Entier_Es.Put(J);
  §  ¨¹¹ Ada.Text_Io.New_Line;
  §  ©end P1;
  §begin
  ¨¹¹ Ada.Text_Io.Put_Line("*** Impression dans Exemple_8, avant P1
***");
  ¨¹¹ Entier_Es.Put(I);   ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ P1;
  ¨¹¹ Ada.Text_Io.Put_Line("*** Impression dans Exemple_8, apres P1
***");
  ¨¹¹ Entier_Es.Put(I);
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ Entier_Es.Put(L);
  ©end Exemple_8;
```

L'exécution de ce programme produit les résultats suivant:

```
*** Impression dans Exemple_8, avant P1 ***
      1
*** Impression dans P1, avant P2 ***
      2
      3
*** Impression dans P2, apres begin ***
      2
      4
*** Impression dans P2, avant end ***
      7
      6
      5
      8
*** Impression dans P1, apres P2 ***
      7
      3
*** Impression dans Exemple_8, apres P1 ***
      1
      8
```

Commenter ces résultats.

9.7 Quelques exercices

9.7.1 Exercice 1

Soit le sous programme suivant:
Tapez le tel quel, que remarque – t – on à la compilations? Expliquer et corriger ce code.

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
ÞßàprocedureExemple_9 is
 ªË¹¹¹¹¹¹¹¹
  §
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Positif_Es is new Ada.Text_Io.Integer_Io(Positive);
  § £¦¦¦¦¦¦¦¦¦
  §
  § ¬¹¹¹¹¹¹¹¹¹
  §ÞßàprocedureEcrire_Triangle_De_Pascal (
  § §                  N : in     Positive ) is
```

```
§ ªË11111111
§  §begin
§  ¨11±for I in 1..N loop
§  §  711±for J in 1..I loop
§  §  5  711 Positif_Es.Put(Coefficient_Du_Triangle_De_Pascal(I,J));
§  §  5  °end loop;
§  §  711 Ada.Text_Io.New_Line;
§  §  °end loop;
§  ©end Ecrire_Triangle_De_Pascal;
§
§ ¬111111111
§Þßàfunction Coefficient_Du_Triangle_De_Pascal (
§ §                   I,
§ §                   J : in     Positive ) return Positive is
§ ªË11111111
§  § í K : Positive;
§  §begin
§  ¨13´if J=1 or J=I then
§  § 6¾11 K:=1;
§  § ¶´else
§  § ¸¾11 K:=Coefficient_Du_Triangle_De_Pascal(I-1,J-1)+
§  § ¸        Coefficient_Du_Triangle_De_Pascal(I-1,J);
§  § È end if;
§Â1Ä11 return K;
§  ©end Coefficient_Du_Triangle_De_Pascal;
§
§ í N : Positive := 5;
§
§begin
¨11 Ecrire_Triangle_De_Pascal(N);
©end Exemple_9;
```

On remarque ici l'utilisation de la récursivité dans la définition de cette fonction

9.7.2 Exercice 2

Soit les sous – programme suivant;

```
 with Ada.Text_Io;
¬111111111
Þßàprocedure Exemple_10 is
ªË11111111
 §
 § ï type T_Element is digits 5;
 § ï type T_Tableau is array (Positive range <>) of T_Element;
 §
 § ¡¥¥¥¥¥¥¥¥
 §ØÓìpackage T_Element_Es is new Ada.Text_Io.Float_Io(T_Element);
 § £¦¦¦¦¦¦¦¦¦
 § ¡¥¥¥¥¥¥¥¥
 §ØÓìpackage Positif_Es is new Ada.Text_Io.Integer_Io(Positive);
 § £¦¦¦¦¦¦¦¦¦
 §
 § ¬111111111
 §Þßàprocedure Lire_Tableau (
 § §                   Taille  : in     Positive  := 3;
 § §                   Tableau :    out T_Tableau       ) is
 § ªË11111111
 §  § í Table : T_Tableau (1 .. Taille);
 §  §begin
 §  ¨11±for I in 1..Taille loop
 §  §  711 T_Element_Es.Get(Table(I));
```

```
§  §  °end loop;
§  ¨¹¹ Tableau:=Table;
§  ©end Lire_Tableau;
§
§ ¬¹¹¹¹¹¹¹¹
§ÞßàfunctionLire_Tableau (
§ §                   Taille : in     Positive := 3 )
§ § return T_Tableau is
§ ªË¹¹¹¹¹¹¹
§  § í Table : T_Tableau (1 .. Taille);
§  §begin
§  ¨¹¹±for I in 1..Taille loop
§  §  7¹¹ T_Element_Es.Get(Table(I));
§  §  °end loop;
§Â¹Ä¹¹ return Table;
§  ©end Lire_Tableau;
§
§ ¬¹¹¹¹¹¹¹¹
§Þßàprocedure Ecrire_Tableau (
§ §                   Tableau : in     T_Tableau ) is
§ ªË¹¹¹¹¹¹¹
§  §begin
§  ¨¹¹±for I in Tableau'range loop
§  §  7¹¹ Ada.Text_Io.Put("tableau (");
§  §  7¹¹ Positif_Es.Put(I);
§  §  7¹¹ Ada.Text_Io.Put(") = ");
§  §  7¹¹ T_Element_Es.Put(Tableau(I));
§  §  7¹¹ Ada.Text_Io.New_Line;
§  §  °end loop;
§  ©end Ecrire_Tableau;
§
§begin
§-----------------------------------------------------------------
§--Ecrire un programme qui illustre l'utilisation
§--de ces sous programmes dans tous les cas possibles
§-----------------------------------------------------------------
©end Exemple_10;
```

9.7.3 Exercice 3

Soit le programme suivant:

```
 with Ada.Text_Io;
¬¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_11 is
ªË¹¹¹¹¹¹¹
§ ¡¥¥¥¥¥¥¥¥¥
§ØÓìpackage Entier_Es is new Ada.Text_Io.Integer_Io(Integer);
§ £¦¦¦¦¦¦¦¦¦
§ í I,
§   J,
§   K : Integer;
§ ¬¹¹¹¹¹¹¹¹
§Þßàprocedure P1 is
§ ªË¹¹¹¹¹¹¹
§  § í I : Integer;
§  §begin
§  ¨¹¹ I:=5;
§  ¨¹¹ Ada.Text_Io.Put("p1: ");
§  ¨¹¹ Entier_Es.Put(I);
```

```
§  ¨¹¹ Entier_Es.Put(J);
§  ¨¹¹ Entier_Es.Put(K);
§  ¨¹¹ Ada.Text_Io.New_Line;
§  ¨¹¹ J:=6;
§  ¨¹¹ K:=7;
§  ©end  P1;
§ ¬¹¹¹¹¹¹¹¹¹
§ÞßàprocedureP2is
§ ªË¹¹¹¹¹¹¹¹
§  § í I,
§  §   J : Integer;
§  § ¬¹¹¹¹¹¹¹¹¹
§  §ÞßàprocedureP3is
§  § ªË¹¹¹¹¹¹¹¹
§  §  § í J,
§  §  §   K : Integer;
§  §  §begin
§  §  ¨¹¹ I:=3;
§  §  ¨¹¹ J:=4;
§  §  ¨¹¹ K:=8;
§  §  ¨¹¹ P1;
§  §  ¨¹¹ Ada.Text_Io.Put("p3: ");
§  §  ¨¹¹ Entier_Es.Put(I);
§  §  ¨¹¹ Entier_Es.Put(J);
§  §  ¨¹¹ Entier_Es.Put(K);
§  §  ¨¹¹ Ada.Text_Io.New_Line;
§  §  ©end P3;
§  §begin
§  ¨¹¹ I:=-1;
§  ¨¹¹ J:=-2;
§  ¨¹¹ K:=9;
§  ¨¹¹ P1;
§  ¨¹¹ Ada.Text_Io.Put("p2: ");
§  ¨¹¹ Entier_Es.Put(I);
§  ¨¹¹ Entier_Es.Put(J);
§  ¨¹¹ Entier_Es.Put(K);
§  ¨¹¹ Ada.Text_Io.New_Line;
§  ¨¹¹ P3;
§  ¨¹¹ Ada.Text_Io.Put("p2: ");
§  ¨¹¹ Entier_Es.Put(I);
§  ¨¹¹ Entier_Es.Put(J);
§  ¨¹¹ Entier_Es.Put(K);
§  ¨¹¹ Ada.Text_Io.New_Line;
§  ©end P2;
§begin
¨¹¹ I:=0;
¨¹¹ J:=1;
¨¹¹ K:=2;
¨¹¹ P1;
¨¹¹ Ada.Text_Io.Put("pr: ");
¨¹¹ Entier_Es.Put(I);
¨¹¹ Entier_Es.Put(J);
¨¹¹ Entier_Es.Put(K);
¨¹¹ Ada.Text_Io.New_Line;
¨¹¹ P2;
¨¹¹ Ada.Text_Io.Put("pr: ");
¨¹¹ Entier_Es.Put(I);
¨¹¹ Entier_Es.Put(J);
¨¹¹ Entier_Es.Put(K);
¨¹¹ Ada.Text_Io.New_Line;
©end Exemple_11;
```

Analyser et commenter les résultats obtenus.

9.7.4 Exercice 4

Soit le programme suivant:

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_12 is
 ªË¹¹¹¹¹¹¹¹
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Entier_Es is new Ada.Text_Io.Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦¦
  § ï type Tableau is array (1 .. 10) of Integer;
  § í T : Tableau;
  § í I : Integer;
  § ¬¹¹¹¹¹¹¹¹¹
  §Þßàprocedure Divise (
  § §                  M,
  § §                  N : in     Integer;
  § §                  Q,
  § §                  R :    out Integer  ) is
  § ªË¹¹¹¹¹¹¹¹
  §  § í Q1,
  §  §   I  : Integer;
  §  §begin
  §  ¨¹¹ Q1:=0;
  §  ¨¹¹ I:=M;
  §  ¨¹¹±while I>=N loop
  §  §  7¹¹ I:=I-N;
  §  §  7¹¹ Q1:=Q1+1;
  §  §  °end loop;
  §  ¨¹¹ Q:=Q1;
  §  ¨¹¹ R:=I;
  §  ©end Divise;
  §begin
  ¨¹¹ I:=1;
  ¨¹¹ Ada.Text_Io.Put("division 1 ");
  ¨¹¹ Ada.Text_Io.New_Line;
  ¨¹¹ Divise(8,3,T(I),T(I+1));
  ¨¹¹ Ada.Text_Io.Put("83");
  ¨¹¹ Entier_Es.Put(T(I));
  ¨¹¹ Entier_Es.Put(T(I+1));
  ©end Exemple_12;
```

Analyser et commenter les résultats obtenus.

9.7.5 Exercice 5

Soit le programme suivant:

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_13 is
 ªË¹¹¹¹¹¹¹¹
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Entier_Es is new Ada.Text_Io.Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦¦
  § ï type Tableau is array (1 .. 10) of Integer;
```

```
§ í T : Tableau;
§ í I : Integer;
§ ¬¹¹¹¹¹¹¹¹¹
§Þßàprocedure Divise (
§ §                  M,
§ §                  N : in     Integer;
§ §                  Q,
§ §                  R :    out Integer  ) is
§ ªË¹¹¹¹¹¹¹¹
§  § í Q1,
§  §   I  : Integer;
§  §begin
§  ¨¹¹ Q1:=0;
§  ¨¹¹ I:=M;
§  ¨¹¹±while I>=N loop
§  §  7¹¹ I:=I-N;
§  §  7¹¹ Q1:=Q1+1;
§  §  °end loop;
§  ¨¹¹ Q:=Q1;
§  ¨¹¹ R:=I;
§  ©end Divise;
§begin
¨¹¹ I:=1;
¨¹¹ Ada.Text_Io.Put("division 2 ");
¨¹¹ Ada.Text_Io.New_Line;
¨¹¹ Divise(8,3,I,T(I));
¨¹¹ Ada.Text_Io.Put("83");
¨¹¹ Entier_Es.Put(I);
¨¹¹ Entier_Es.Put(T(I));
©end Exemple_13;
```

Analyser et commenter les résultats obtenus.

9.7.6 Exercice 6

Soit le programme suivant:

```
  with Ada.Text_Io;
¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_14 is
ªË¹¹¹¹¹¹¹¹
§ ¡¥¥¥¥¥¥¥¥
§ØÓìpackage Entier_Es is new Ada.Text_Io.Integer_Io(Integer);
§ £¦¦¦¦¦¦¦¦¦
§ í I,
§   J : Integer;
§ ¬¹¹¹¹¹¹¹¹¹
§Þßàfunction F (
§ §           I : in     Integer )
§ § return Integer is
§ ªË¹¹¹¹¹¹¹¹
§  §begin
§Â¹Ä¹¹ return I*(I-1)/2;
§  ©end F;
§ ¬¹¹¹¹¹¹¹¹¹
§Þßàprocedure Q is
§ ªË¹¹¹¹¹¹¹¹
§  §begin
§  ¨¹¹ I:=F(I);
§  ¨¹¹ I:=I/2;
```

```
§  ¨¹¹ Ada.Text_Io.Put("Q : ");
§  ¨¹¹ Entier_Es.Put(I);
§  ¨¹¹ Entier_Es.Put(J);
§  ¨¹¹ Ada.Text_Io.New_Line;
§  ©end Q;
§ ¬¹¹¹¹¹¹¹¹¹
§Þßàprocedure P (
§ §            Y : in     Integer;
§ §            I : in out Integer  ) is
§ ªË¹¹¹¹¹¹¹¹
§  §begin
§  ¨¹¹ Ada.Text_Io.Put("P : ");
§  ¨¹¹ Entier_Es.Put(I);
§  ¨¹¹ Entier_Es.Put(J);
§  ¨¹¹ Ada.Text_Io.New_Line;
§  ¨¹¹ Q;
§  ¨¹¹ I:=Y+J;
§  ¨¹¹ J:=F(I);
§  ¨¹¹ Ada.Text_Io.Put("E : ");
§  ¨¹¹ Entier_Es.Put(I);
§  ¨¹¹ Entier_Es.Put(J);
§  ¨¹¹ Ada.Text_Io.New_Line;
§  ©end P;
§begin
¨¹¹ I:=2;
¨¹¹ J:=5;
¨¹¹ Ada.Text_Io.Put("D : ");
¨¹¹ Entier_Es.Put(I);
¨¹¹ Entier_Es.Put(J);
¨¹¹ Ada.Text_Io.New_Line;
¨¹¹ P(F(I),J);
¨¹¹ Ada.Text_Io.Put("PP: ");
¨¹¹ Entier_Es.Put(I);
¨¹¹ Entier_Es.Put(J);
©end Exemple_14;
```

Analyser et commenter les résultats obtenus.

9.7.7 Exercice 7

Soit le code Ada suivant:

```
 with Ada.Text_Io;
¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_15 is
ªË¹¹¹¹¹¹¹¹
 § í Longueur_Max : constant := 80;
 § ï type T_Element is new String
 §      (1 .. Longueur_Max);
 § ï type T_Chaine is
 §      record
 §         La_Chaine : T_Element;
 §         Longueur  : Natural;
 §      end record;
 §
 § ¬¹¹¹¹¹¹¹¹¹
 §Þßàfunction Lire_T_Chaine return T_Chaine is
 § ªË¹¹¹¹¹¹¹¹
 §  § í S        : T_Chaine  := ((others => ' '), 0);
 §  § í Compteur : Natural;
```

```
  §  § í C        : Character;
  §  §begin
  §  ¨¹¹ Compteur:=0;
  §  ¨¹¹±while not Ada.Text_Io.End_Of_Line loop
  §  §  7¹¹ Ada.Text_Io.Get(C);
  §  §  7¹¹ Compteur:=Compteur+1;
  §  §  7¹¹ S.La_Chaine(Compteur):=C;
  §  §  °end loop;
  §  ¨¹¹ S.Longueur:=Compteur;
  §Â¹Ä¹¹ return S;
  §  § ¬¹¹¹¹¹¹¹¹¹
  §  §ðîìexception
  §  § ªË¹¹¹¹¹¹¹¹
  §  §  ¨¹³´when Constraint_Error =>
  §  §  § 6¨¹¹ Ada.Text_Io.Put("*** la chaine est tronquée a :
"&Integer
  §  §  § 6§                   'Image(T_Element'Last)&" caracteres !
" );
  §  §  § 6¨¹¹ Ada.Text_Io.New_Line;
  §  §  § 6¨¹¹ S.Longueur:=T_Element'Last;
  §Â¹Ä  © ¶¾¹¹ return S;
  §  ©end Lire_T_Chaine;
  §--------------------------------------------------------------------
  §-- Ecrire une procédure qui imprime à l'écran
  §-- un objet de type T_Chaine
  §--------------------------------------------------------------------
  §
  §--------------------------------------------------------------------
  §-- Ecrire une fonction qui compte le nombre de mot
  §-- un objet de type T_Chaine
  §--------------------------------------------------------------------
  §
  §--------------------------------------------------------------------
  §-- Ecrire une procedure qui extrait et imprime à l'écran
  §-- les mots d'un objet de type T_Chaine
  §--------------------------------------------------------------------
  §
  §--------------------------------------------------------------------
  §-- Ecrire une procedure qui extrait le premier mot
  §-- d'un objet de type T_Chaine, et retourne sa valeur
  §-- dans un objet de type T_Chaine
  §--------------------------------------------------------------------
  §
  §begin
  §--------------------------------------------------------------------
  §-- Ecrire une programme qui illustre l'utilisation de
  §-- ces sous - programmes
  §--------------------------------------------------------------------
  ©end Exemple_15;
```

10 Les Fichier

10.1 Introduction

Un fichier est une collection d'informations de même nature, rangées sur un support physique différent de la mémoire centrale. Autrement dit, un fichier est une structure organisée permettant de stocker et de restituer des informations.

On distingue plusieurs types de fichiers, suivant les informations à stocker et les moyens d'accéder à ces informations.

Le support physique est en général le disque dur ou la disquette ou le CD-ROM, …

L'objectif général de l'utilisation des fichiers est donc l'enregistrement et la récupération des informations.

10.2 Différents types de fichiers

Les différents types de fichiers sont multiples et variés. Ceci dit, on peut séparer une catégorie composée des fichiers textes. Ces fichiers sont constitués de suites de caractères mais sont peu esthétiques puisqu'il n'y a pas de format, de police, de style, … Les fichiers non textes sont façonnables à loisir et dépendent de la structure des informations que vous y mettez.

10.3 Utilisations

Les fichiers sont utiles dès que l'on veut stocker des informations de manière permanente (même si l'application n'est pas en cours d'exécution). On les rencontre dans tous les types d'application : fichiers bureautiques, fichiers textes, fichiers binaires, fichiers bases de données, …

10.4 Modes d'organisation

10.4.1 Généralités

Il existe deux modes d'organisation principaux des fichiers ; attention rien à voir avec le type d'informations stockées.

Les fichiers à organisation *séquentielle*, où toutes les informations sont accessibles l'une après l'autre à partir de la première. On accède donc aux données dans l'ordre dans lequel elles sont rangées. La longueur des informations est variable. Ce type d'organisation convient bien à tous les supports physiques. Ce mode d'organisation est similaire à la notion de liste chaînée.

Les fichiers à organisation *accès direct*, où toutes les informations sont directement accessibles par leur rang (indice), c'est-à-dire, leur positionnement par rapport à la première information qui a pour rang (indice) la valeur 1. La longueur des informations est fixe. Ce mode d'organisation est similaire à la notion de tableau.

10.4.2 Avantages et inconvénients pour chacun de ces modes d'organisation

Mode d'organisation		
Séquentiel	Nombre d'informations limité que par la taille du support physique, Adapté à tous les types de supports.	Accès lent, du fait de la lecture séquentielle.
Direct	Accès rapide, accès direct à l'information recherchée.	Nombre d'informations définies au préalable.

Un fichier séquentiel peut être schématisé par un ruban défilant devant une tête de lecture / écriture. On n'a accès à une valeur que lorsque l'information correspondante (élément courant) est en face de la tête. Un fichier étant une suite finie, on introduit une marque sur le ruban qui permet au mécanisme d'interprétation de détecter la fin du fichier. Un fichier n'est défini que s'il comporte une marque.

10.5 Fichier logique – fichier physique

Dans un ordinateur, tout fichier est identifié :

- au niveau du système d'exploitation par un *nom physique* qui est une chaîne de caractères,

- au niveau du langage de programmation par un identificateur *(nom logique*), comme toute autre variable.

L'association de l'identificateur à un fichier physique de nom externe donné, se fait le plus souvent à l'ouverture (choix du mode d'utilisation), ou en séparant l'association de l'ouverture au moyen d'une instruction spécifique d'association.

Une fois l'utilisation du fichier terminée, il ne faut pas oublier de rompre l'association, c'est-à-dire, de fermer le fichier.

10.6 Modes d'utilisation

Les différents modes d'utilisation sont la récupération, le stockage et la modification des informations. Le mode d'utilisation dépend de l'organisation du fichier en ce qui concerne notamment la modification des informations.

Mode d'utilisation	Organisation séquentielle	Organisation accès direct
Récupération (Lecture)	Limite l'utilisation du fichier aux seules lectures des informations	Limite l'utilisation du fichier aux seules lectures des informations
Stockage (Ecriture)	Permet la création d'un fichier. Autorise l'écriture des informations dans le fichier	Permet la création d'un fichier. Autorise l'écriture des informations dans le fichier
Modification	Permet l'ajout d'information à la fin d'un fichier existant. La modification proprement dite nécessite une « recopie » par un algorithme à programmer.	Permet des lectures et/ou écritures des informations.

On dit que l'on *ouvre* le fichier pour un mode d'utilisation.

Par exemple, pour un fichier séquentiel avec comme mode d'utilisation la lecture, l'ouverture du fichier dans ce mode rend possible l'accès à la première information du fichier si elle existe. Elle positionne le ruban de telle sorte que la tête soit en face de la première information du fichier pour en permettre la lecture. Un prédicat (présenté ci-dessous) de fin de fichier prend la valeur vrai si la tête est positionnée en face de la marque de fin de fichier. (Un fichier peut être vide).

Toujours pour un fichier séquentiel, l'ouverture en écriture permet la création d'un nouveau fichier vide, et le prédicat de fin de fichier prend la valeur vraie.

10.7 Opérations et prédicats

10.7.1 Prédicat

Fin de Fichier

Ce prédicat (fonction booléenne) prend la valeur vrai si on a atteint la marque de fin de fichier ; il prend la valeur faux dans le cas contraire.

C'est l'exécution de l'ouverture dans un mode d'utilisation et des opérations de lecture et d'écriture qui permet de donner une valeur au prédicat de fin de fichier.

10.7.2 Opérations

Une fois le lien réalisé entre le fichier physique (externe) et le fichier logique (variable), il faut ouvrir le fichier dans le mode d'utilisation adéquat. Ces deux opérations, association et ouverture, sont réalisées au sein d'une même instruction suivant les langages.

Une fois l'ouverture réalisée, les différentes opérations possibles sont la lecture et / ou l'écriture des informations depuis ou vers le fichier.

Lecture

La lecture, lire(nom_de_fichier, val), pour une organisation séquentielle provoque dans l'ordre :

- la *copie* de l'élément courant pour la mettre dans la variable val,
- l'*avancement* du curseur d'une position.

Ecriture

L'écriture sera notée écrire(nom_de_fichier, val).

On ne peut utiliser cette primitive que si la tête est positionnée en fin de fichier (fin de fichier à vrai). L'interprétation de cette primitive est la suivante ; on effectue dans l'ordre

- la *recopie*, en fin de fichier, de la valeur contenue dans la variable val,
- l'*avancement* du curseur d'une position afin de positionner la tête sur la marque de fin de fichier.

D'autres prédicats et opérations existent suivant les types de fichiers et / ou leur mode d'organisation. Par exemple, pour des fichiers séquentiels de type texte, on peut rajouter un prédicat permettant de détecter une fin de ligne. Pour des fichiers en accès direct, on rajoutera des primitives permettant de se positionner à l'indice i…

10.8 Les fichiers en Ada

10.8.1 Les fichiers séquentiels

Le fichier est vu comme une suite d'informations de même nature. La nature (le type) gérée dans les fichiers séquentiels est souvent un type article (ou enregistrement).

Pour les fichiers séquentiels, les entrées-sorties de valeurs sont définies au moyen du paquetage générique SEQUENTIAL_IO.

```
   with Ada.IO_Exceptions;
 ¡¥ÍÍÍÍÍÍÍÍ
ØÙÚgeneric
 ¢ type Element_Type(<>) is private;
 ¢ package Ada.Sequential_IO is
 £¤ÎÎÎÎÎÎÎÎ
  §
  § ï type File_Type is limited private;
  §
  § ï type File_Mode is (In_File, Out_File, Append_File);
  §
  §-- File management
  §
  § ¬111111111
  §ÛÜÝprocedure Create(File : in out File_Type;
  § §                  Mode : in File_Mode := Out_File;
  § §                  Name : in String := "";
  § §                  Form : in String := "");
  § ª111111111
  §
  § ¬111111111
  §ÛÜÝprocedure Open (File : in out File_Type;
  § §                 Mode : in File_Mode;
  § §                 Name : in String;
  § §                 Form : in String := "");
  § ª111111111
  §
  § ¬111111111
  §ÛÜÝprocedure Close (File : in out File_Type);
  § ª111111111
  § ¬111111111
```

```
  §ÛÜÝprocedure Delete(File : in out File_Type);
  § ª111111111
  § ¬111111111
  §ÛÜÝprocedure Reset (File : in out File_Type; Mode : in File_Mode);
  § ª111111111
  § ¬111111111
  §ÛÜÝprocedure Reset (File : in out File_Type);
  § ª111111111
  §
  § ¬111111111
  §ÛÜÝfunction Mode (File : in File_Type) return File_Mode;
  § ª111111111
  § ¬111111111
  §ÛÜÝfunction Name (File : in File_Type) return String;
  § ª111111111
  § ¬111111111
  §ÛÜÝfunction Form (File : in File_Type) return String;
  § ª111111111
  §
  § ¬111111111
  §ÛÜÝfunction Is_Open(File : in File_Type) return Boolean;
  § ª111111111
  §
  §-- Input and output operations
  §
  § ¬111111111
  §ÛÜÝprocedure Read (File : in File_Type; Item : out Element_Type);
  § ª111111111
  § ¬111111111
  §ÛÜÝprocedure Write (File : in File_Type; Item : in Element_Type);
  § ª111111111
  §
  § ¬111111111
  §ÛÜÝfunction End_Of_File(File : in File_Type) return Boolean;
  § ª111111111
  §
  §-- Exceptions
  §
  § í Status_Error : exception renames IO_Exceptions.Status_Error;
  § í Mode_Error : exception renames IO_Exceptions.Mode_Error;
  § í Name_Error : exception renames IO_Exceptions.Name_Error;
  § í Use_Error : exception renames IO_Exceptions.Use_Error;
  § í Device_Error : exception renames IO_Exceptions.Device_Error;
  § í End_Error : exception renames IO_Exceptions.End_Error;
  § í Data_Error : exception renames IO_Exceptions.Data_Error;
  §
  §private
  §-- not specified by the language
  ©end Ada.Sequential_IO;
```

Pour définir les entrées-sorties séquentielles pour un type d'éléments concerné, déclarez une instanciation de cette unité générique, avec le type donné comme paramètre générique effectif.

```
  ¡¥¥¥¥¥¥¥¥
ØÓìpackage P_SEQ_IO is new ADA.SEQUENTIAL_IO (T_Element);
 £¦¦¦¦¦¦¦¦¦
```

Mode d'utilisation

Les modes autorisés pour les fichiers séquentiels sont les modes en lecture (IN_FILE), en écriture (OUT_FILE), et en ajout (APPEND_FILE).

Déclaration d'un fichier

Les fichiers séquentiels sont des objets d'un type FILE_TYPE déclaré par instanciation de SEQUENTIAL_IO. Ils sont déclarés de la façon suivante :

```
í FichierSequentiel1 : P_SEQ_IO.FILE_TYPE ;
```

FichierSequentiel1 est le nom interne du fichier.

Ouverture et fermeture d'un fichier

Ouvrir un fichier est une instruction qui précède dans l'algorithme l'utilisation du fichier. Pour cela, on dispose dans le paquetage SEQUENTIAL_IO des deux procédures :

```
¬111111111
ÛÜÝprocedure Create(File : in out File_Type;
 §                  Mode : in File_Mode := Out_File;
 §                  Name : in String := "";
 §                  Form : in String := "");
 a111111111

¬111111111
ÛÜÝprocedure Open (File : in out File_Type;
 §                 Mode : in File_Mode;
 §                 Name : in String;
 §                 Form : in String := "");
 a111111111
```

La procédure CREATE ouvre un fichier en vue de sa création, mode écriture obligatoire, et la procédure OPEN ouvre un fichier qui existe déjà (MODE au choix) soit donc en lecture, soit en écriture (mais l'ancien est perdu) ou en ajout. Si le fichier existe déjà, l'ouverture avec CREATE le supprime. Si le fichier n'existe pas, l'ouverture avec OPEN lève une exception. En supplément de l'ouverture, l'association entre le nom physique et le nom logique du fichier est faite grâce à CREATE ou OPEN.

Le paramètre FORM est fourni pour pouvoir spécifier des informations auxiliaires liées à l'implémentation ; sa valeur par défaut est vide et son utilisation facultative.

```
CREATE (FichierSequentiel1, NAME => "MonFichier" ) ;
OPEN (FichierSequentiel1, IN_FILE, "MonFichier" ) ;
```

La procédure RESET repositionne le fichier, ouvert avec OPEN, de telle sorte que la lecture de ces éléments puisse reprendre au début du fichier. Si le paramètre MODE est précisé, le mode courant du fichier est positionné au mode donné, cela permet de changer de mode.

```
¬111111111
ÛÜÝprocedure Reset (File : in out File_Type; Mode : in File_Mode);
 a111111111
¬111111111
ÛÜÝprocedure Reset (File : in out File_Type);
 a111111111
```

La procédure DELETE détruit le fichier physique associé au fichier donné. Le fichier spécifié est fermé et le fichier physique cesse d'exister ; c'est utile pour les fichiers dits temporaires. A bien maîtriser.

```
¬111111111
ÛÜÝprocedure Delete(File : in out File_Type);
 a111111111
```

La fermeture du fichier est réalisée par l'intermédiaire de la procédure CLOSE. Elle détruit l'association entre les deux noms de fichier (logique et physique) dans les trois modes.

```
 ¬111111111
ÛÜÝprocedure Close (File : in out File_Type);
 a111111111
```

10.8.2 Prédicats et Opérations

Le prédicat qui permet de savoir si on se trouve en fin de fichier est le suivant :

```
 ¬111111111
ÛÜÝfunction End_Of_File(File : in File_Type) return Boolean;
 a111111111
```

On dispose aussi d'un prédicat qui permet de savoir si un fichier est ouvert, c'est-à-dire, s'il est associé à un fichier externe : IS_OPEN. On peut connaître le nom du fichier physique associé au fichier donné par la fonction : NAME.

```
 ¬111111111
ÛÜÝfunction Is_Open(File : in File_Type) return Boolean;
 a111111111
 ¬111111111
ÛÜÝfunction Name (File : in File_Type) return String;
 a111111111
```

Les opérations disponibles pour la lecture et l'écriture d'un élément sont :

```
 ¬111111111
ÛÜÝprocedure Read (File : in File_Type; Item : out Element_Type);
 a111111111
 ¬111111111
ÛÜÝprocedure Write (File : in File_Type; Item : in Element_Type);
 a111111111
```

La procédure READ agit sur le fichier en mode IN_FILE. Elle lit un élément du fichier et retourne la valeur de cet élément dans le paramètre résultat ITEM.

La procédure WRITE agit sur un fichier en mode OUT_FILE ou APPEND_FILE. Elle écrit la valeur de ITEM dans le fichier.

10.8.3 Traitement des exceptions

Il existe un paquetage IO_EXCEPTIONS qui définit les exceptions requises dans SEQUENTIAL_IO, DIRECT_IO, et TEXT_IO.

Extraits du manuel de référence annexe A.13 Exceptions in Input-Output

Manuel de référence A.13 Exceptions in Input-Output

The package IO_Exceptions defines the exceptions needed by the predefined input-output packages.

The library package IO_Exceptions has the following declaration:

```
 ¡¥¥¥¥¥¥¥¥
ØÓìpackage Ada.Io_Exceptions is
 £¤¦¦¦¦¦¦¦
  §pragma Pure(Io_Exceptions);
  § í Status_Error : exception;
  § í Mode_Error   : exception;
  § í Name_Error   : exception;
  § í Use_Error    : exception;
  § í Device_Error : exception;
  § í End_Error    : exception;
  § í Data_Error   : exception;
  § í Layout_Error : exception;
  ©end Ada.Io_Exceptions;
```

If more than one error condition exists, the corresponding exception that appears earliest in the following list is the one that is propagated.

The exception Status_Error is propagated by an attempt to operate upon a file that is not open, and by an attempt to open a file that is already open.

The exception Mode_Error is propagated by an attempt to read from, or test for the end of, a file whose current mode is Out_File or Append_File, and also by an attempt to write to a file whose current mode is In_File. In the case of Text_IO, the exception Mode_Error is also propagated by specifying a file whose current mode is Out_File or Append_File in a call of Set_Input, Skip_Line, End_Of_Line, Skip_Page, or End_Of_Page; and by specifying a file whose current mode is In_File in a call of Set_Output, Set_Line_Length, Set_Page_Length, Line_Length, Page_Length, New_Line, or New_Page.

The exception Name_Error is propagated by a call of Create or Open if the string given for the parameter Name does not allow the identification of an external file. For example, this exception is propagated if the string is improper, or, alternatively, if either none or more than one external file corresponds to the string.

The exception Use_Error is propagated if an operation is attempted that is not possible for reasons that depend on characteristics of the external file. For example, this exception is propagated by the procedure Create, among other circumstances, if the given mode is Out_File but the form specifies an input only device, if the parameter Form specifies invalid access rights, or if an external file with the given name already exists and overwriting is not allowed.

The exception Device_Error is propagated if an input-output operation cannot be completed because of a malfunction of the underlying system.

The exception End_Error is propagated by an attempt to skip (read past) the end of a file.

The exception Data_Error can be propagated by the procedure Read (or by the Read attribute) if the element read cannot be interpreted as a value of the required subtype. This exception is also propagated by a procedure Get (defined in the package Text_IO) if the input character sequence fails to satisfy the required syntax, or if the value input does not belong to the range of the required subtype.

The exception Layout_Error is propagated (in text input-output) by Col, Line, or Page if the value returned exceeds Count'Last. The exception Layout_Error is also propagated on output by an attempt to set column or line numbers in excess of specified maximum line or page lengths, respectively (excluding the unbounded cases). It is also propagated by an attempt to Put too many characters to a string.

10.9 Exemples

```
   with Ada.Sequential_Io;
 ¬111111111
Þßàprocedure Exemple_1 is
```

```
 ªË¹¹¹¹¹¹¹¹
  §
  § ï type T_Element is range 1 .. 10;
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Fichier_Element is new  Ada.Sequential_Io(T_Element);
  § £¦¦¦¦¦¦¦¦¦
  §
  § í Fichier : Fichier_Element.File_Type;
  §begin
  ¨¹¹ Fichier_Element.Create(Fichier,Fichier_Element.Out_File,
  §                          "exemple_1.dat");
  ¨¹¹±for I in T_Element'range loop
  §  7¹¹ Fichier_Element.Write(Fichier,I);
  §  °end loop;
  ¨¹¹ Fichier_Element.Close(Fichier);
  ©end Exemple_1;

   with Ada.Text_Io;
   with Ada.Sequential_Io;
 ¬¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_2 is
 ªË¹¹¹¹¹¹¹¹
  §
  § ï type T_Element is range 1 .. 10;
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage T_Element_Es is new Ada.Text_Io.Integer_Io(T_Element);
  § £¦¦¦¦¦¦¦¦¦
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Fichier_Element is new  Ada.Sequential_Io(T_Element);
  § £¦¦¦¦¦¦¦¦¦
  §
  § í Fichier : Fichier_Element.File_Type;
  § í Element : T_Element;
  §begin
  ¨¹¹ Fichier_Element.Open(Fichier,Fichier_Element.In_File,
  §                        "exemple_1.dat");
  ¨¹¹±while not Fichier_Element.End_Of_File(Fichier) loop
  §  7¹¹ Fichier_Element.Read(Fichier,Element);
  §  7¹¹ T_Element_Es.Put(Element);
  §  7¹¹ Ada.Text_Io.New_Line;
  §  °end loop;
  ¨¹¹ Fichier_Element.Close(Fichier);
  ©end Exemple_2;
```

Reprendre le programme `Exemple_2` en changeant le nom du fichier exemple_1.dat en exemple_2.dat.

```
   with Ada.Text_Io;
   with Ada.Sequential_Io;
 ¬¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_3 is
 ªË¹¹¹¹¹¹¹¹
  §
  § ï type T_Element is new Integer;
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage T_Element_Es is new Ada.Text_Io.Integer_Io(T_Element);
  § £¦¦¦¦¦¦¦¦¦
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Fichier_Element is new  Ada.Sequential_Io(T_Element);
  § £¦¦¦¦¦¦¦¦¦
  §
```

```
§ í Fichier        : Fichier_Element.File_Type;
§ í Mouvement      : Fichier_Element.File_Type;
§ í Element        : T_Element;
§ í Element_Insere : T_Element                := - 9999;
§begin
¨¹¹ Fichier_Element.Create(Fichier,Fichier_Element.Out_File,
§                          "exemple_3.dat");
¨¹¹±for I in 1..10 loop
§  7¹¹ Fichier_Element.Write(Fichier,T_Element(I));
§  °end loop;
¨¹¹ Fichier_Element.Close(Fichier);
¨¹¹ Fichier_Element.Open(Fichier,Fichier_Element.In_File,
§                        "exemple_3.dat");
¨¹¹±while not Fichier_Element.End_Of_File(Fichier) loop
§  7¹¹ Fichier_Element.Read(Fichier,Element);
§  7¹¹ T_Element_Es.Put(Element);
§  7¹¹ Ada.Text_Io.New_Line;
§  °end loop;
¨¹¹ Fichier_Element.Reset(Fichier);
¨¹¹ Fichier_Element.Create(Mouvement,Fichier_Element.Out_File,
§                          "mouvement.dat");
¨¹¹±while not Fichier_Element.End_Of_File(Fichier) loop
§  7¹¹ Fichier_Element.Read(Fichier,Element);
§  7¹¹ Fichier_Element.Write(Mouvement,Element);
§  °end loop;
¨¹¹ Fichier_Element.Write(Mouvement,Element_Insere);
¨¹¹ Fichier_Element.Close(Mouvement);
¨¹¹ Fichier_Element.Close(Fichier);
¨¹¹ Fichier_Element.Open(Fichier,Fichier_Element.Out_File,
§                        "exemple_3.dat");
¨¹¹ Fichier_Element.Open(Mouvement,Fichier_Element.In_File,
§                        "mouvement.dat");
¨¹¹±while not Fichier_Element.End_Of_File(Mouvement) loop
§  7¹¹ Fichier_Element.Read(Mouvement,Element);
§  7¹¹ Fichier_Element.Write(Fichier,Element);
§  °end loop;
¨¹¹ Fichier_Element.Delete(Mouvement);
¨¹¹ Fichier_Element.Reset(Fichier,Fichier_Element.In_File);
¨¹¹±while not Fichier_Element.End_Of_File(Fichier) loop
§  7¹¹ Fichier_Element.Read(Fichier,Element);
§  7¹¹ T_Element_Es.Put(Element);
§  7¹¹ Ada.Text_Io.New_Line;
§  °end loop;
©end Exemple_3;
```

10.10Les fichiers textes

Un fichier texte est un objet structuré. Du point de vue logique, un fichier texte est une suite de caractères, de lignes, et éventuellement de pages. Une ligne est une liste, éventuellement vide, de caractères terminée par la marque « fin_de_ligne ». Une page est une liste de lignes, éventuellement vide, terminée par la marque « fin_de_page », et un fichier est une liste de pages, éventuellement vide, terminé par une marque « fin_de_fichier ».

10.10.1 Généralités

Mode d'utilisation

Les modes autorisés pour les fichiers textes sont les mode en lecture (IN_FILE), en écriture (OUT_FILE), et en ajout (APPEND_FILE).

Déclaration d'un fichier

Les fichiers textes sont des objets du type limité privé FILE_TYPE exporté du paquetage ADA.TEXT_IO. Ils sont déclarés de la façon suivante :

```
FichierRuban : ADA.TEXT_IO.FILE_TYPE ;
```

Ouverture et fermeture d'un fichier

Les procédures CREATE, OPEN, DELETE, RESET, et CLOSE sont identiques à celles présentées précédemment.

10.10.2 Prédicats et Opérations

Extraits du manuel de référence annexe A.10.1 The Package Text_IO

```
   with Ada.IO_Exceptions;
 ¡¥¥¥¥¥¥¥¥
ØÓìpackage Ada.Text_IO is
 £¤¦¦¦¦¦¦¦¦
  §
  § ï type File_Type is limited private;
  §
  § ï type File_Mode is (In_File, Out_File, Append_File);
  §
  § ï type Count is range 0 .. implementation-defined;
  § ï subtype Positive_Count is Count range 1 .. Count'Last;
  § í Unbounded : constant Count := 0; -- line and page length
  §
  § ï subtype Field is Integer range 0 .. implementation-defined;
  § ï subtype Number_Base is Integer range 2 .. 16;
  §
  § ï type Type_Set is (Lower_Case, Upper_Case);
  §
  §-- File Management
  §
  § ¬¹¹¹¹¹¹¹¹¹
  §ÛÜÝprocedure Create (File : in out File_Type;
  § §                  Mode : in File_Mode := Out_File;
  § §                  Name : in String := "";
  § §                  Form : in String := "");
  § ª¹¹¹¹¹¹¹¹¹
  §
  § ¬¹¹¹¹¹¹¹¹¹
  §ÛÜÝprocedure Open (File : in out File_Type;
  § §                Mode : in File_Mode;
  § §                Name : in String;
  § §                Form : in String := "");
  § ª¹¹¹¹¹¹¹¹¹
  §
  § ¬¹¹¹¹¹¹¹¹¹
  §ÛÜÝprocedure Close (File : in out File_Type);
  § ª¹¹¹¹¹¹¹¹¹
  § ¬¹¹¹¹¹¹¹¹¹
  §ÛÜÝprocedure Delete (File : in out File_Type);
  § ª¹¹¹¹¹¹¹¹¹
  § ¬¹¹¹¹¹¹¹¹¹
  §ÛÜÝprocedure Reset (File : in out File_Type; Mode : in File_Mode);
  § ª¹¹¹¹¹¹¹¹¹
  § ¬¹¹¹¹¹¹¹¹¹
```

```
§ÛÜÝprocedure Reset (File : in out File_Type);
§ ª111111111
§
§ ¬111111111
§ÛÜÝfunction Mode (File : in File_Type) return File_Mode;
§ ª111111111
§ ¬111111111
§ÛÜÝfunction Name (File : in File_Type) return String;
§ ª111111111
§ ¬111111111
§ÛÜÝfunction Form (File : in File_Type) return String;
§ ª111111111
§
§ ¬111111111
§ÛÜÝfunction Is_Open(File : in File_Type) return Boolean;
§ ª111111111
§
§-- Control of default input and output files
§
§ ¬111111111
§ÛÜÝprocedure Set_Input (File : in File_Type);
§ ª111111111
§ ¬111111111
§ÛÜÝprocedure Set_Output(File : in File_Type);
§ ª111111111
§ ¬111111111
§ÛÜÝprocedure Set_Error (File : in File_Type);
§ ª111111111
§
§ ¬111111111
§ÛÜÝfunction Standard_Input return File_Type;
§ ª111111111
§ ¬111111111
§ÛÜÝfunction Standard_Output return File_Type;
§ ª111111111
§ ¬111111111
§ÛÜÝfunction Standard_Error return File_Type;
§ ª111111111
§
§ ¬111111111
§ÛÜÝfunction Current_Input return File_Type;
§ ª111111111
§ ¬111111111
§ÛÜÝfunction Current_Output return File_Type;
§ ª111111111
§ ¬111111111
§ÛÜÝfunction Current_Error return File_Type;
§ ª111111111
§
§ ï type File_Access is access constant File_Type;
§
§ ¬111111111
§ÛÜÝfunction Standard_Input return File_Access;
§ ª111111111
§ ¬111111111
§ÛÜÝfunction Standard_Output return File_Access;
§ ª111111111
§ ¬111111111
§ÛÜÝfunction Standard_Error return File_Access;
§ ª111111111
§
§ ¬111111111
```

```
§ÛÜÝfunction Current_Input return File_Access;
§ ª111111111
§ ¬111111111
§ÛÜÝfunction Current_Output return File_Access;
§ ª111111111
§ ¬111111111
§ÛÜÝfunction Current_Error return File_Access;
§ ª111111111
§
§-- Buffer control
§ ¬111111111
§ÛÜÝprocedure Flush (File : in out File_Type);
§ ª111111111
§ ¬111111111
§ÛÜÝprocedure Flush;
§ ª111111111
§
§-- Specification of line and page lengths
§
§ ¬111111111
§ÛÜÝprocedure Set_Line_Length(File : in File_Type; To : in Count);
§ ª111111111
§ ¬111111111
§ÛÜÝprocedure Set_Line_Length(To : in Count);
§ ª111111111
§
§ ¬111111111
§ÛÜÝprocedure Set_Page_Length(File : in File_Type; To : in Count);
§ ª111111111
§ ¬111111111
§ÛÜÝprocedure Set_Page_Length(To : in Count);
§ ª111111111
§
§ ¬111111111
§ÛÜÝfunction Line_Length(File : in File_Type) return Count;
§ ª111111111
§ ¬111111111
§ÛÜÝfunction Line_Length return Count;
§ ª111111111
§
§ ¬111111111
§ÛÜÝfunction Page_Length(File : in File_Type) return Count;
§ ª111111111
§ ¬111111111
§ÛÜÝfunction Page_Length return Count;
§ ª111111111
§
§-- Column, Line, and Page Control
§
§ ¬111111111
§ÛÜÝprocedure New_Line (File : in File_Type;
§ §                      Spacing : in Positive_Count := 1);
§ ª111111111
§ ¬111111111
§ÛÜÝprocedure New_Line (Spacing : in Positive_Count := 1);
§ ª111111111
§
§ ¬111111111
§ÛÜÝprocedure Skip_Line (File : in File_Type;
§ §                       Spacing : in Positive_Count := 1);
§ ª111111111
§ ¬111111111
```

```
§ÛÜÝprocedure Skip_Line (Spacing : in Positive_Count := 1);
§ ª111111111
§
§ ¬111111111
§ÛÜÝfunction End_Of_Line(File : in File_Type) return Boolean;
§ ª111111111
§ ¬111111111
§ÛÜÝfunction End_Of_Line return Boolean;
§ ª111111111
§
§ ¬111111111
§ÛÜÝprocedure New_Page (File : in File_Type);
§ ª111111111
§ ¬111111111
§ÛÜÝprocedure New_Page;
§ ª111111111
§
§ ¬111111111
§ÛÜÝprocedure Skip_Page (File : in File_Type);
§ ª111111111
§ ¬111111111
§ÛÜÝprocedure Skip_Page;
§ ª111111111
§
§ ¬111111111
§ÛÜÝfunction End_Of_Page(File : in File_Type) return Boolean;
§ ª111111111
§ ¬111111111
§ÛÜÝfunction End_Of_Page return Boolean;
§ ª111111111
§
§ ¬111111111
§ÛÜÝfunction End_Of_File(File : in File_Type) return Boolean;
§ ª111111111
§ ¬111111111
§ÛÜÝfunction End_Of_File return Boolean;
§ ª111111111
§
§ ¬111111111
§ÛÜÝprocedure Set_Col (File : in File_Type; To : in Positive_Count);
§ ª111111111
§ ¬111111111
§ÛÜÝprocedure Set_Col (To : in Positive_Count);
§ ª111111111
§
§ ¬111111111
§ÛÜÝprocedure Set_Line(File : in File_Type; To : in Positive_Count);
§ ª111111111
§ ¬111111111
§ÛÜÝprocedure Set_Line(To : in Positive_Count);
§ ª111111111
§
§ ¬111111111
§ÛÜÝfunction Col (File : in File_Type) return Positive_Count;
§ ª111111111
§ ¬111111111
§ÛÜÝfunction Col return Positive_Count;
§ ª111111111
§
§ ¬111111111
§ÛÜÝfunction Line(File : in File_Type) return Positive_Count;
§ ª111111111
```

```
§ ¬¹¹¹¹¹¹¹¹¹
§ÛÜÝfunction Line return Positive_Count;
§ ª¹¹¹¹¹¹¹¹¹
§
§ ¬¹¹¹¹¹¹¹¹¹
§ÛÜÝfunction Page(File : in File_Type) return Positive_Count;
§ ª¹¹¹¹¹¹¹¹¹
§ ¬¹¹¹¹¹¹¹¹¹
§ÛÜÝfunction Page return Positive_Count;
§ ª¹¹¹¹¹¹¹¹¹
§
§-- Character Input-Output
§
§ ¬¹¹¹¹¹¹¹¹¹
§ÛÜÝprocedure Get(File : in File_Type; Item : out Character);
§ ª¹¹¹¹¹¹¹¹¹
§ ¬¹¹¹¹¹¹¹¹¹
§ÛÜÝprocedure Get(Item : out Character);
§ ª¹¹¹¹¹¹¹¹¹
§
§ ¬¹¹¹¹¹¹¹¹¹
§ÛÜÝprocedure Put(File : in File_Type; Item : in Character);
§ ª¹¹¹¹¹¹¹¹¹
§ ¬¹¹¹¹¹¹¹¹¹
§ÛÜÝprocedure Put(Item : in Character);
§ ª¹¹¹¹¹¹¹¹¹
§
§ ¬¹¹¹¹¹¹¹¹¹
§ÛÜÝprocedure Look_Ahead (File : in File_Type;
§ §                        Item : out Character;
§ §                        End_Of_Line : out Boolean);
§ ª¹¹¹¹¹¹¹¹¹
§ ¬¹¹¹¹¹¹¹¹¹
§ÛÜÝprocedure Look_Ahead (Item : out Character;
§ §                        End_Of_Line : out Boolean);
§ ª¹¹¹¹¹¹¹¹¹
§
§ ¬¹¹¹¹¹¹¹¹¹
§ÛÜÝprocedure Get_Immediate(File : in File_Type;
§ §                         Item : out Character);
§ ª¹¹¹¹¹¹¹¹¹
§ ¬¹¹¹¹¹¹¹¹¹
§ÛÜÝprocedure Get_Immediate(Item : out Character);
§ ª¹¹¹¹¹¹¹¹¹
§
§ ¬¹¹¹¹¹¹¹¹¹
§ÛÜÝprocedure Get_Immediate(File : in File_Type;
§ §                         Item : out Character;
§ §                         Available : out Boolean);
§ ª¹¹¹¹¹¹¹¹¹
§ ¬¹¹¹¹¹¹¹¹¹
§ÛÜÝprocedure Get_Immediate(Item : out Character;
§ §                         Available : out Boolean);
§ ª¹¹¹¹¹¹¹¹¹
§
§-- String Input-Output
§
§ ¬¹¹¹¹¹¹¹¹¹
§ÛÜÝprocedure Get(File : in File_Type; Item : out String);
§ ª¹¹¹¹¹¹¹¹¹
§ ¬¹¹¹¹¹¹¹¹¹
§ÛÜÝprocedure Get(Item : out String);
```

```
§ ª111111111
§
§ ¬111111111
§ÛÜÝprocedure Put(File : in File_Type; Item : in String);
§ ª111111111
§ ¬111111111
§ÛÜÝprocedure Put(Item : in String);
§ ª111111111
§
§ ¬111111111
§ÛÜÝprocedure Get_Line(File : in File_Type;
§ §                    Item : out String;
§ §                    Last : out Natural);
§ ª111111111
§ ¬111111111
§ÛÜÝprocedure Get_Line(Item : out String; Last : out Natural);
§ ª111111111
§
§ ¬111111111
§ÛÜÝprocedure Put_Line(File : in File_Type; Item : in String);
§ ª111111111
§ ¬111111111
§ÛÜÝprocedure Put_Line(Item : in String);
§ ª111111111
§
§-- Generic packages for Input-Output of Integer Types
§
§ ¡¥ÍÍÍÍÍÍÍÍ
§ØÙÚgeneric
§ ¢ type Num is range <>;
§ ¢ package Integer_IO is
§ £¤ÎÎÎÎÎÎÎÎ
§  §
§  § í Default_Width : Field := Num'Width;
§  § í Default_Base : Number_Base := 10;
§  §
§  § ¬111111111
§  §ÛÜÝprocedure Get(File : in File_Type;
§  § §               Item : out Num;
§  § §               Width : in Field := 0);
§  § ª111111111
§  § ¬111111111
§  §ÛÜÝprocedure Get(Item : out Num;
§  § §               Width : in Field := 0);
§  § ª111111111
§  §
§  § ¬111111111
§  §ÛÜÝprocedure Put(File : in File_Type;
§  § §               Item : in Num;
§  § §               Width : in Field := Default_Width;
§  § §               Base : in Number_Base := Default_Base);
§  § ª111111111
§  § ¬111111111
§  §ÛÜÝprocedure Put(Item : in Num;
§  § §               Width : in Field := Default_Width;
§  § §               Base : in Number_Base := Default_Base);
§  § ª111111111
§  § ¬111111111
§  §ÛÜÝprocedure Get(From : in String;
§  § §               Item : out Num;
§  § §               Last : out Positive);
§  § ª111111111
```

```
§  § ¬¹¹¹¹¹¹¹¹¹
§  §ÛÜÝprocedure Put(To : out String;
§  § §               Item : in Num;
§  § §               Base : in Number_Base := Default_Base);
§  § ª¹¹¹¹¹¹¹¹¹
§  §
§  ©end Integer_IO;
§
§ ¡¥ÍÍÍÍÍÍÍÍ
§ØÙÚgeneric
§ ¢ type Num is mod <>;
§ ¢ package Modular_IO is
§ £¤ÎÎÎÎÎÎÎÎ
§  §
§  § í Default_Width : Field := Num'Width;
§  § í Default_Base : Number_Base := 10;
§  §
§  § ¬¹¹¹¹¹¹¹¹¹
§  §ÛÜÝprocedure Get(File : in File_Type;
§  § §               Item : out Num;
§  § §               Width : in Field := 0);
§  § ª¹¹¹¹¹¹¹¹¹
§  § ¬¹¹¹¹¹¹¹¹¹
§  §ÛÜÝprocedure Get(Item : out Num;
§  § §               Width : in Field := 0);
§  § ª¹¹¹¹¹¹¹¹¹
§  §
§  § ¬¹¹¹¹¹¹¹¹¹
§  §ÛÜÝprocedure Put(File : in File_Type;
§  § §               Item : in Num;
§  § §               Width : in Field := Default_Width;
§  § §               Base : in Number_Base := Default_Base);
§  § ª¹¹¹¹¹¹¹¹¹
§  § ¬¹¹¹¹¹¹¹¹¹
§  §ÛÜÝprocedure Put(Item : in Num;
§  § §               Width : in Field := Default_Width;
§  § §               Base : in Number_Base := Default_Base);
§  § ª¹¹¹¹¹¹¹¹¹
§  § ¬¹¹¹¹¹¹¹¹¹
§  §ÛÜÝprocedure Get(From : in String;
§  § §               Item : out Num;
§  § §               Last : out Positive);
§  § ª¹¹¹¹¹¹¹¹¹
§  § ¬¹¹¹¹¹¹¹¹¹
§  §ÛÜÝprocedure Put(To : out String;
§  § §               Item : in Num;
§  § §               Base : in Number_Base := Default_Base);
§  § ª¹¹¹¹¹¹¹¹¹
§  §
§  ©end Modular_IO;
§
§-- Generic packages for Input-Output of Real Types
§
§ ¡¥ÍÍÍÍÍÍÍÍ
§ØÙÚgeneric
§ ¢ type Num is digits <>;
§ ¢ package Float_IO is
§ £¤ÎÎÎÎÎÎÎÎ
§  §
§  § í Default_Fore : Field := 2;
§  § í Default_Aft : Field := Num'Digits-1;
§  § í Default_Exp : Field := 3;
```

```
§  §
§  § ¬¹¹¹¹¹¹¹¹¹
§  §ÛÜÝprocedure Get(File : in File_Type;
§  § §               Item : out Num;
§  § §               Width : in Field := 0);
§  § ª¹¹¹¹¹¹¹¹¹
§  § ¬¹¹¹¹¹¹¹¹¹
§  §ÛÜÝprocedure Get(Item : out Num;
§  § §               Width : in Field := 0);
§  § ª¹¹¹¹¹¹¹¹¹
§  §
§  § ¬¹¹¹¹¹¹¹¹¹
§  §ÛÜÝprocedure Put(File : in File_Type;
§  § §               Item : in Num;
§  § §               Fore : in Field := Default_Fore;
§  § §               Aft : in Field := Default_Aft;
§  § §               Exp : in Field := Default_Exp);
§  § ª¹¹¹¹¹¹¹¹¹
§  § ¬¹¹¹¹¹¹¹¹¹
§  §ÛÜÝprocedure Put(Item : in Num;
§  § §               Fore : in Field := Default_Fore;
§  § §               Aft : in Field := Default_Aft;
§  § §               Exp : in Field := Default_Exp);
§  § ª¹¹¹¹¹¹¹¹¹
§  §
§  § ¬¹¹¹¹¹¹¹¹¹
§  §ÛÜÝprocedure Get(From : in String;
§  § §               Item : out Num;
§  § §               Last : out Positive);
§  § ª¹¹¹¹¹¹¹¹¹
§  § ¬¹¹¹¹¹¹¹¹¹
§  §ÛÜÝprocedure Put(To : out String;
§  § §               Item : in Num;
§  § §               Aft : in Field := Default_Aft;
§  § §               Exp : in Field := Default_Exp);
§  § ª¹¹¹¹¹¹¹¹¹
§  ©end Float_IO;
§
§ ¡¥ÍÍÍÍÍÍÍÍ
§ØÙÚgeneric
§ ¢ type Num is delta <>;
§ ¢ package Fixed_IO is
§ £¤ÎÎÎÎÎÎÎÎ
§  §
§  § í Default_Fore : Field := Num'Fore;
§  § í Default_Aft : Field := Num'Aft;
§  § í Default_Exp : Field := 0;
§  §
§  § ¬¹¹¹¹¹¹¹¹¹
§  §ÛÜÝprocedure Get(File : in File_Type;
§  § §               Item : out Num;
§  § §               Width : in Field := 0);
§  § ª¹¹¹¹¹¹¹¹¹
§  § ¬¹¹¹¹¹¹¹¹¹
§  §ÛÜÝprocedure Get(Item : out Num;
§  § §               Width : in Field := 0);
§  § ª¹¹¹¹¹¹¹¹¹
§  §
§  § ¬¹¹¹¹¹¹¹¹¹
§  §ÛÜÝprocedure Put(File : in File_Type;
§  § §               Item : in Num;
§  § §               Fore : in Field := Default_Fore;
```

```
§  § §               Aft : in Field := Default_Aft;
§  § §               Exp : in Field := Default_Exp);
§  § ª111111111
§  § ¬111111111
§  §ÛÜÝprocedure Put(Item : in Num;
§  § §               Fore : in Field := Default_Fore;
§  § §               Aft : in Field := Default_Aft;
§  § §               Exp : in Field := Default_Exp);
§  § ª111111111
§  §
§  § ¬111111111
§  §ÛÜÝprocedure Get(From : in String;
§  § §               Item : out Num;
§  § §               Last : out Positive);
§  § ª111111111
§  § ¬111111111
§  §ÛÜÝprocedure Put(To : out String;
§  § §               Item : in Num;
§  § §               Aft : in Field := Default_Aft;
§  § §               Exp : in Field := Default_Exp);
§  § ª111111111
§  ©end Fixed_IO;
§
§ ¡¥ÍÍÍÍÍÍÍÍ
§ØÙÚgeneric
§ ¢ type Num is delta <> digits <>;
§ ¢ package Decimal_IO is
§ £¤ÎÎÎÎÎÎÎÎ
§  §
§  § í Default_Fore : Field := Num'Fore;
§  § í Default_Aft : Field := Num'Aft;
§  § í Default_Exp : Field := 0;
§  §
§  § ¬111111111
§  §ÛÜÝprocedure Get(File : in File_Type;
§  § §               Item : out Num;
§  § §               Width : in Field := 0);
§  § ª111111111
§  § ¬111111111
§  §ÛÜÝprocedure Get(Item : out Num;
§  § §               Width : in Field := 0);
§  § ª111111111
§  §
§  § ¬111111111
§  §ÛÜÝprocedure Put(File : in File_Type;
§  § §               Item : in Num;
§  § §               Fore : in Field := Default_Fore;
§  § §               Aft : in Field := Default_Aft;
§  § §               Exp : in Field := Default_Exp);
§  § ª111111111
§  § ¬111111111
§  §ÛÜÝprocedure Put(Item : in Num;
§  § §               Fore : in Field := Default_Fore;
§  § §               Aft : in Field := Default_Aft;
§  § §               Exp : in Field := Default_Exp);
§  § ª111111111
§  §
§  § ¬111111111
§  §ÛÜÝprocedure Get(From : in String;
§  § §               Item : out Num;
§  § §               Last : out Positive);
§  § ª111111111
```

```
§  § ¬111111111
§  §ÛÜÝprocedure Put(To : out String;
§  § §               Item : in Num;
§  § §               Aft : in Field := Default_Aft;
§  § §               Exp : in Field := Default_Exp);
§  § ª111111111
§  ©end Decimal_IO;
§
§-- Generic package for Input-Output of Enumeration Types
§
§ ¡¥ÍÍÍÍÍÍÍÍ
§ØÙÚgeneric
§ ¢ type Enum is (<>);
§ ¢ package Enumeration_IO is
§ £¤ÎÎÎÎÎÎÎÎ
§  §
§  § í Default_Width : Field := 0;
§  § í Default_Setting : Type_Set := Upper_Case;
§  §
§  § ¬111111111
§  §ÛÜÝprocedure Get(File : in File_Type;
§  § §               Item : out Enum);
§  § ª111111111
§  § ¬111111111
§  §ÛÜÝprocedure Get(Item : out Enum);
§  § ª111111111
§  §
§  § ¬111111111
§  §ÛÜÝprocedure Put(File : in File_Type;
§  § §               Item : in Enum;
§  § §               Width : in Field := Default_Width;
§  § §               Set : in Type_Set := Default_Setting);
§  § ª111111111
§  § ¬111111111
§  §ÛÜÝprocedure Put(Item : in Enum;
§  § §               Width : in Field := Default_Width;
§  § §               Set : in Type_Set := Default_Setting);
§  § ª111111111
§  §
§  § ¬111111111
§  §ÛÜÝprocedure Get(From : in String;
§  § §               Item : out Enum;
§  § §               Last : out Positive);
§  § ª111111111
§  § ¬111111111
§  §ÛÜÝprocedure Put(To : out String;
§  § §               Item : in Enum;
§  § §               Set : in Type_Set := Default_Setting);
§  § ª111111111
§  ©end Enumeration_IO;
§
§-- Exceptions
§
§ í Status_Error : exception renames IO_Exceptions.Status_Error;
§ í Mode_Error : exception renames IO_Exceptions.Mode_Error;
§ í Name_Error : exception renames IO_Exceptions.Name_Error;
§ í Use_Error : exception renames IO_Exceptions.Use_Error;
§ í Device_Error : exception renames IO_Exceptions.Device_Error;
§ í End_Error : exception renames IO_Exceptions.End_Error;
§ í Data_Error : exception renames IO_Exceptions.Data_Error;
§ í Layout_Error : exception renames IO_Exceptions.Layout_Error;
§private
```

```
  §-- not specified by the language
  ©end Ada.Text_IO;
```

Les procédures GET et PUT tiennent à jour les numéros courants de page, de colonne et de ligne du fichier spécifié. GET s'utilise sur un fichier en mode IN_FILE et PUT sur les autres modes.

Le clavier (en lecture) et l'écran (en écriture) jouent un rôle particulier. Ils sont considérés comme des fichiers textes (dits standard) ayant les identificateurs implicites : STANDARD_INPUT et STANDARD_OUTPUT. Ces identificateurs sont pris par défaut si le nom du fichier est omis dans les procédures et les fonctions de lecture/écriture. Les notions des TP précédents n'étaient donc qu'un cas particulier de ce TP sur les fichiers.

On ne peut pas ouvrir, fermer, repositionner, ou détruire les fichiers d'entrée et de sortie standard.

10.11Entrées_sorties pour les types entiers (signés ou modulaires).

Les procédures suivantes sont définies, pour les entiers signés, dans le paquetage générique INTEGER_IO inclus dans le paquetage ADA.TEXT_IO ou pour les entiers modulaires dans le sous-paquetage générique MODULAR_IO.

Le sous-paquetage générique INTEGER_IO doit être instancié pour le type entier approprié, indiqué par le paramètre formel NUM dans les procédures.

10.12Entrées_sorties pour les types réels (flottants ou fixes ou décimaux).

Les procédures utiles pour les entrées-sorties sont définies dans les paquetages génériques FLOAT_IO, FIXED_IO ou DECIMAL_IO inclus dans le paquetage ADA.TEXT_IO.

Se reporter au manuel de référence pour avoir plus d'information.

10.13Entrées_sorties pour les types énumératifs.

Les procédures utiles pour les entrées-sorties sont définies dans le paquetage générique ENUMERATION_IO inclu dans le paquetage ADA.TEXT_IO.

VSe reporter au manuel de référence pour avoir plus d'information.

10.14Exemples

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_4 is
 ªË¹¹¹¹¹¹¹¹
  §
  § ¡¥¥¥¥¥¥¥¥
  §ØÓìpackage Entier_Es is new Ada.Text_Io.Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦¦
  §
  § í I        : Integer              := 123456;
  § í Chaine   : String (1 .. 80);
  § í Longueur : Natural;
  § í Fichier  : Ada.Text_Io.File_Type;
  §begin
  ¨¹¹ Ada.Text_Io.Create(Fichier,Ada.Text_Io.Out_File,
  §                      "exemple_4.txt");
  ¨¹¹ Ada.Text_Io.Set_Line_Length(Fichier,40);
  ¨¹¹ Ada.Text_Io.Get_Line(Chaine,Longueur);
  ¨¹¹ Ada.Text_Io.Put(Fichier,Chaine(1..Longueur)&" "
  §                   &Integer'Image(I));
  ¨¹¹ Ada.Text_Io.Close(Fichier);
  ©end Exemple_4;
```

```
   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_5 is
 ªË¹¹¹¹¹¹¹¹
  §
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Entier_Es is new Ada.Text_Io.Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦¦
  §
  § í Caractere : Character;
  § í Fichier   : Ada.Text_Io.File_Type;
  §begin
  ¨¹¹ Ada.Text_Io.Open(Fichier,Ada.Text_Io.In_File,
  §                    "exemple_4.txt");
  ¨¹¹±while not Ada.Text_Io.End_Of_File(Fichier) loop
  §  7¹¹ Ada.Text_Io.Get(Fichier,Caractere);
  §  7¹¹ Ada.Text_Io.Put(Caractere);
  §  °end loop;
  ¨¹¹ Ada.Text_Io.Close(Fichier);
  ©end Exemple_5;

   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_6 is
 ªË¹¹¹¹¹¹¹¹
  §
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Entier_Es is new Ada.Text_Io.Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦¦
  §
  § í Caractere : Character;
  § í Fichier   : Ada.Text_Io.File_Type;
  §begin
  ¨¹¹ Ada.Text_Io.Open(Fichier,Ada.Text_Io.In_File,
  §                    "exemple_4.txt");
  ¨¹¹±while not Ada.Text_Io.End_Of_File(Fichier) loop
  §  7¹³´if Ada.Text_Io.End_Of_Line(Fichier) then
  §  5 6¨¹¹ Ada.Text_Io.Skip_Line(Fichier);
  §  5 6¾¹¹ Ada.Text_Io.New_Line;
  §  5 ¶´else
  §  5 ¸¨¹¹ Ada.Text_Io.Get(Fichier,Caractere);
  §  5 ¸¾¹¹ Ada.Text_Io.Put(Caractere);
  §  5 È end if;
  §  °end loop;
  ¨¹¹ Ada.Text_Io.Close(Fichier);
  ©end Exemple_6;

   with Ada.Text_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_7 is
 ªË¹¹¹¹¹¹¹¹
  §
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Entier_Es is new Ada.Text_Io.Integer_Io(Integer);
  § £¦¦¦¦¦¦¦¦¦
  §
  § í Caractere_Look_Ahead : Character;
  § í Caractere            : Character;
  § í Fin_De_Ligne         : Boolean;
  § í I                    : Integer;
  § í Fichier              : Ada.Text_Io.File_Type;
```

```
§begin
¨¹¹ Ada.Text_Io.Open(Fichier,Ada.Text_Io.In_File,
§                    "exemple_4.txt");
¨¹¹±while not Ada.Text_Io.End_Of_File(Fichier) loop
§  7¹³´if Ada.Text_Io.End_Of_Line(Fichier) then
§  5 6¨¹¹ Ada.Text_Io.Skip_Line(Fichier);
§  5 6¾¹¹ Ada.Text_Io.New_Line;
§  5 ¶´else
§  5 ¸¨¹¹ Ada.Text_Io.Look_Ahead(Fichier,Caractere_Look_Ahead,
§  5 ¸§                          Fin_De_Ligne);
§  5 ¸¾¹³´if Caractere_Look_Ahead >= '0' and
§  5 ¸  6§Caractere_Look_Ahead <= '9' then
§  5 ¸  6¨¹¹ Entier_Es.Get(Fichier,I);
§  5 ¸  6¾¹¹ Entier_Es.Put(I);
§  5 ¸  ¶´else
§  5 ¸  ¸¨¹¹ Ada.Text_Io.Get(Fichier,Caractere);
§  5 ¸  ¸¾¹¹ Ada.Text_Io.Put(Caractere);
§  5 ¸  È end if;
§  5 È end if;
§  °end loop;
¨¹¹ Ada.Text_Io.Close(Fichier);
©end Exemple_7;

 with Ada.Text_Io;
¬¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_8 is
ªË¹¹¹¹¹¹¹¹
§
§ ¡¥¥¥¥¥¥¥¥¥
§ØÓìpackage Entier_Es is new Ada.Text_Io.Integer_Io(Integer);
§ £¦¦¦¦¦¦¦¦¦
§
§ ¬¹¹¹¹¹¹¹¹
§Þßàfunction Longueur_Entier (
§ §                  A : in     Integer )
§ § return Natural is
§ ªË¹¹¹¹¹¹¹
§  § í Chaine   : String (1 .. Entier_Es.Default_Width) :=
§  §      (others => ' ');
§  § í Compteur : Natural                              := 0;
§  §begin
§  ¨¹¹ Entier_Es.Put(Chaine,A);
§  ¨¹¹±for I in 1..Entier_Es.Default_Width loop
§  §  7¹³´if Chaine(I)=' ' then
§  §  5 6¾¹¹ Compteur:=Compteur+1;
§  §  5 ¶ end if;
§  §  °end loop;
§Â¹Ä¹¹ return Entier_Es.Default_Width-Compteur;
§  ©end Longueur_Entier;
§
§ í Caractere_Look_Ahead : Character;
§ í Caractere            : Character;
§ í Fin_De_Ligne         : Boolean;
§ í I                    : Integer;
§ í Fichier              : Ada.Text_Io.File_Type;
§begin
¨¹¹ Ada.Text_Io.Open(Fichier,Ada.Text_Io.In_File,
§                    "exemple_4.txt");
¨¹¹±while not Ada.Text_Io.End_Of_File(Fichier) loop
§  7¹³´if Ada.Text_Io.End_Of_Line(Fichier) then
§  5 6¨¹¹ Ada.Text_Io.Skip_Line(Fichier);
§  5 6¾¹¹ Ada.Text_Io.New_Line;
```

```
§  5 ¶´else
§  5 ¸¨¹¹ Ada.Text_Io.Look_Ahead(Fichier,Caractere_Look_Ahead,
§  5 ¸§                          Fin_De_Ligne);
§  5 ¸¾¹³´if Caractere_Look_Ahead >= '0' and
§  5 ¸  6§Caractere_Look_Ahead <= '9' then
§  5 ¸  6¨¹¹ Entier_Es.Get(Fichier,I);
§  5 ¸  6¾¹¹ Entier_Es.Put(
§  5 ¸  6                Item  => I,
§  5 ¸  6                Width => Longueur_Entier (I));
§  5 ¸  ¶´else
§  5 ¸  ¸¨¹¹ Ada.Text_Io.Get(Fichier,Caractere);
§  5 ¸  ¸¾¹¹ Ada.Text_Io.Put(Caractere);
§  5 ¸  È end if;
§  5 È end if;
§  °end loop;
¨¹¹ Ada.Text_Io.Close(Fichier);
©end Exemple_8;
```

10.15Les fichiers à accès direct.

Le fichier est vu comme un ensemble d'informations de même nature occupant des positions consécutives en ordre linéaire. Une valeur peut être transférée de, ou vers, un élément du fichier situé à n'importe quelle position choisie. La position d'un élément est spécifiée par un indice (rang, ou index) qui est un nombre de type COUNT. Le premier élément a pour indice 1.

Un fichier à accès direct gère un indice courant. Cet indice courant est utilisé par la prochaine opération de lecture ou d'écriture. A l'ouverture l'indice courant est implicitement positionné à 1. Cet indice courant est la propriété de l'objet fichier logique. Il est géré automatiquement par les opérations d'entrées-sorties mais aussi par quelques instructions spéciales mises en œuvre par le programmeur.

Pour ces fichiers, les entrées-sorties de valeurs, d'un même type d'élément, sont définies au moyen du paquetage générique DIRECT_IO. Le paramètre générique n'est pas à discriminant comme cela était possible avec les fichiers séquentiels.

Pour utiliser les entrées-sorties directes pour un type d 'élément donné, déclarez une instanciation de cette unité générique avec le type concerné comme paramètre générique effectif.

10.15.1 Mode d'utilisation

Les modes autorisés pour les fichiers à accès direct sont les modes en lecture (IN_FILE), en écriture (OUT_FILE), et en modification (INOUT_FILE). Le mode INOUT_FILE permet des lectures et / ou écritures, dans le même algorithme, sur un même fichier. Le mode APPEND_FILE n'existe plus.

10.15.2 Déclaration d'un fichier

Les fichiers à accès direct sont des objets d'un type FILE_TYPE déclaré par instanciation de DIRECT_IO. Ils sont déclarés de la façon suivante :

```
FichierAccesDirect : P_DIR_IO.FILE_TYPE ;
```

10.15.3 Ouverture et fermeture d'un fichier

Les procédures CREATE, OPEN, DELETE, RESET, et CLOSE sont identiques à leurs homologues de gestion de fichier séquentiel.

10.15.4 Prédicats et Opérations

Les prédicats END_OF_FILE, IS_OPEN, MODE, et NAME sont identiques à leurs homologues de gestion de fichier séquentiel. On ajoute cependant les prédicats suivants :

```
--    retourne la valeur de l'indice courant du fichier donné
```

```
function INDEX ( FILE : in FILE_TYPE ) return POSITIVE_COUNT;
--      retourne la taille courante du fichier physique associé au fichier
--logique
function SIZE ( FILE : in FILE_TYPE ) return COUNT;
```

Les opérations disponibles pour la lecture pour les fichiers ouverts en mode IN_FILE et INOUT_FILE sont :

```
procedure READ (    FILE    : in FILE_TYPE ;
                                                ITEM    : out ELEMENT_TYPE
;
                                                FROM    : in
POSITIVE_COUNT ) ;
procedure READ (    FILE    : in FILE_TYPE ;
                                                ITEM    : out
ELEMENT_TYPE );
```

La première forme de l'instruction READ agit sur un fichier en mode IN_FILE ou INOUT_FILE tandis que la dernière agit sur un fichier en mode IN_FILE seulement.

La première forme de l'instruction READ positionne l'indice courant à la valeur donnée par le paramètre FROM puis retourne dans le paramètre ITEM la valeur de l 'élément dont la position dans le fichier donné est spécifiée par l'indice courant. Cette procédure READ peut alterner avec WRITE dans l'algorithme.

La deuxième forme de l'instruction READ travaille avec la valeur courante de l'indice courant, qu'elle gère elle-même et retourne dans le paramètre ITEM la valeur de l'élément situé à cet endroit.

Les opérations disponibles pour l 'écriture pour les fichiers ouverts en mode OUT_FILE et INOUT_FILE sont :

```
procedure WRITE (    FILE    : in FILE_TYPE ;
                                                ITEM    : in
ELEMENT_TYPE
                                                TO              : in
POSITIVE_COUNT ) ;
procedure WRITE (    FILE    : in FILE_TYPE ;
                                                ITEM    : in
ELEMENT_TYPE ) ;
```

La première forme de l'instruction WRITE positionne l'indice courant à la valeur donnée par le paramètre TO puis, pour les deux formes, écrit dans le fichier donné, la valeur du paramètre ITEM à la position spécifiée par l'indice courant, celui-ci est utilisé en l'état dans la deuxième forme.

Chaque opération READ ou WRITE se termine en augmentant l'indice courant d'une position, prêt à lire ou à écrire l'enregistrement suivant. Donc attention à la séquence : lecture – modification – écriture, car sans gestion de l'indice courant, vous écrirez la modification sur l'enregistrement suivant et non sur l'enregistrement à modifier. Avant la réécriture, il faut initialiser à nouveau l'indice à la valeur de la lecture, il est prudent de le mémoriser.

On dispose, en outre de la procédure supplémentaire suivante :

```
--      agit sur un fichier de mode quelconque. Positionne l'indice courant
--du fichier à la valeur de l'indice donné par le paramètre TO
procedure SET_INDEX ( FILE : in FILE_TYPE ; TO : in POSITIVE_COUNT );
```

10.16Exemples

```
   with Ada.Direct_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_9 is
 ªË¹¹¹¹¹¹¹¹
  §
  § ï type T_Element is range 1 .. 10;
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Fichier_Element is new  Ada.Direct_Io(T_Element);
  § £¦¦¦¦¦¦¦¦¦
  §
  § í Fichier : Fichier_Element.File_Type;
  §begin
  ¨¹¹ Fichier_Element.Create(Fichier,Fichier_Element.Out_File,
  §                          "exemple_9.dat");
  ¨¹¹±for I in T_Element'range loop
  §  7¹¹ Fichier_Element.Write(Fichier,I);
  §  °end loop;
  ¨¹¹ Fichier_Element.Close(Fichier);
  ©end Exemple_9;

   with Ada.Text_Io;
   with Ada.Direct_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_10 is
 ªË¹¹¹¹¹¹¹¹
  §
  § ï type T_Element is range 1 .. 10;
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Element_Es is new Ada.Text_Io.Integer_Io(T_Element);
  § £¦¦¦¦¦¦¦¦¦
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Fichier_Element is new  Ada.Direct_Io(T_Element);
  § £¦¦¦¦¦¦¦¦¦
  §
  § í Fichier : Fichier_Element.File_Type;
  § í Element : T_Element;
  §begin
  ¨¹¹ Fichier_Element.Open(Fichier,Fichier_Element.In_File,
  §                        "exemple_9.dat");
  ¨¹¹±for I in T_Element'range loop
  §  7¹¹ Fichier_Element.Read(Fichier,Element);
  §  7¹¹ Element_Es.Put(Element);
  §  °end loop;
  ¨¹¹ Fichier_Element.Close(Fichier);
  ©end Exemple_10;

   with Ada.Text_Io;
   with Ada.Direct_Io;
 ¬¹¹¹¹¹¹¹¹¹
Þßàprocedure Exemple_11 is
 ªË¹¹¹¹¹¹¹¹
  §
  § ï type T_Element is range 1 .. 10;
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Element_Es is new Ada.Text_Io.Integer_Io(T_Element);
  § £¦¦¦¦¦¦¦¦¦
  § ¡¥¥¥¥¥¥¥¥¥
  §ØÓìpackage Fichier_Element is new  Ada.Direct_Io(T_Element);
  § £¦¦¦¦¦¦¦¦¦
```

```
§ ¡¥¥¥¥¥¥¥¥
§ØÓìpackage Positive_Count_Es is new
§ ¢ Ada.Text_Io.Integer_Io(Fichier_Element.Positive_Count);
§ £¦¦¦¦¦¦¦¦¦
§   use type Fichier_Element.Positive_Count;
§
§ í Fichier  : Fichier_Element.File_Type;
§ í Element  : T_Element;
§ í Position : Fichier_Element.Positive_Count;
§begin
¨¹¹ Fichier_Element.Open(Fichier,Fichier_Element.Inout_File,
§                        "exemple_9.dat");
¨¹¹ Position:=3;
¨¹¹ Fichier_Element.Read(Fichier,Element,Position);
¨¹¹ Ada.Text_Io.Put("L'element a la position ");
¨¹¹ Positive_Count_Es.Put(Position);
¨¹¹ Ada.Text_Io.Put(" est ");
¨¹¹ Element_Es.Put(Element);
¨¹¹ Ada.Text_Io.New_Line;
¨¹¹ Position:=Position+2;
¨¹¹ Fichier_Element.Read(Fichier,Element,Position);
¨¹¹ Ada.Text_Io.Put("L'element a la position ");
¨¹¹ Positive_Count_Es.Put(Position);
¨¹¹ Ada.Text_Io.Put(" est ");
¨¹¹ Element_Es.Put(Element);
¨¹¹ Ada.Text_Io.New_Line;
¨¹¹ Fichier_Element.Write(Fichier,9,Position);
¨¹¹ Fichier_Element.Read(Fichier,Element,Position);
¨¹¹ Ada.Text_Io.Put("L'element a la position ");
¨¹¹ Positive_Count_Es.Put(Position);
¨¹¹ Ada.Text_Io.Put(" est ");
¨¹¹ Element_Es.Put(Element);
¨¹¹ Fichier_Element.Close(Fichier);
©end Exemple_11;
```

10.17Exercice

Pour illustrer ces notions, il vous est demandé de réaliser une application permettant la gestion des élèves de l'IG2I, en vue d'établir des listes d'élèves par promotion. Les informations connues pour chaque élève sont son nom, son prénom, sa promotion et son groupe.

10.18Quelques exemples supplémentaires

```
   with Ada.Streams.Stream_Io;
 ¬¹11111111
Þßàprocedure Exemple_12 is
 ªË¹1111111
  §
  § í Flot_In  : Ada.Streams.Stream_Io.File_Type;
  § í Flot_Out : Ada.Streams.Stream_Io.File_Type;
  § í Element  : Ada.Streams.Stream_Element_Array (1 .. 1);
  § í Last     : Ada.Streams.Stream_Element_Offset;
  §
  §begin
  ¨¹¹ Ada.Streams.Stream_Io.Open(Flot_In,
  §                              Ada.Streams.Stream_Io.In_File      ,
  §                              "loco.jpg");
  ¨¹¹ Ada.Streams.Stream_Io.Create(Flot_Out,
  §                                Ada.Streams.Stream_Io.Out_File,
  §                                "loco2.jpg");
```

```
   ¨¹¹±while not Ada.Streams.Stream_Io.End_Of_File(Flot_In) loop
   §  7¹¹ Ada.Streams.Stream_Io.Read(Flot_In,Element,Last);
   §  7¹¹ Ada.Streams.Stream_Io.Write(Flot_Out,Element);
   §  °end loop;
   ¨¹¹ Ada.Streams.Stream_Io.Close(Flot_In);
   ¨¹¹ Ada.Streams.Stream_Io.Close(Flot_Out);
   ©end Exemple_12;

    with Ada.Streams.Stream_Io;
  ¬¹11111111
 Þßàprocedure Exemple_13 is
  ªË¹1111111
   §
   § í Flot_In         : Ada.Streams.Stream_Io.File_Type;
   § í Flot_Out        : Ada.Streams.Stream_Io.File_Type;
   § í Ptr_De_Flot_In  : Ada.Streams.Stream_Io.Stream_Access;
   § í Ptr_De_Flot_Out : Ada.Streams.Stream_Io.Stream_Access;
   § í Element         : Ada.Streams.Stream_Element;
   §
   §begin
   ¨¹¹ Ada.Streams.Stream_Io.Open(Flot_In,
   §                              Ada.Streams.Stream_Io.In_File,
   §                              "loco.jpg");
   ¨¹¹ Ada.Streams.Stream_Io.Create(Flot_Out,
   §                                Ada.Streams.Stream_Io.Out_File,
   §                                "loco2.jpg");
   ¨¹¹ Ptr_De_Flot_In:=Ada.Streams.Stream_Io.Stream(Flot_In);
   ¨¹¹ Ptr_De_Flot_Out:=Ada.Streams.Stream_Io.Stream(Flot_Out);
   ¨¹¹±while not Ada.Streams.Stream_Io.End_Of_File(Flot_In) loop
   §  7¹¹ Ada.Streams.Stream_Element'Read(Ptr_De_Flot_In,Element);
   §  7¹¹ Ada.Streams.Stream_Element'Write(Ptr_De_Flot_Out,Element);
   §  °end loop;
   ¨¹¹ Ada.Streams.Stream_Io.Close(Flot_In);
   ¨¹¹ Ada.Streams.Stream_Io.Close(Flot_Out);
   ©end Exemple_13;

    with Ada.Text_Io;
    with Ada.Streams.Stream_Io;
  ¬¹11111111
 Þßàprocedure Exemple_14 is
  ªË¹1111111
   §
   § ¡¥¥¥¥¥¥¥¥
   §ØÓìpackage Size_Es is new
   § ¢ Ada.Text_Io.Integer_Io(Ada.Streams.Stream_Io.Count);
   § £¦¦¦¦¦¦¦¦¦
   §
   § ï type T_Pointeur is access Ada.Streams.Stream_Element_Array;
   § í Flot_In  : Ada.Streams.Stream_Io.File_Type;
   § í Flot_Out : Ada.Streams.Stream_Io.File_Type;
   § í Taille   : Ada.Streams.Stream_Io.Count;
   § í Pointeur : T_Pointeur;
   §begin
   ¨¹¹ Ada.Streams.Stream_Io.Open(Flot_In,
   §                              Ada.Streams.Stream_Io.In_File,
   §                              "loco.jpg");
   ¨¹¹ Taille:=Ada.Streams.Stream_Io.Size(Flot_In);
   ¨¹¹ Ada.Text_Io.Put("La taille du fichier est : ");
   ¨¹¹ Size_Es.Put(Taille);
   ¨¹¹ Ada.Text_Io.Put(" octet(s)");
   ¨¹¹ Ada.Text_Io.New_Line;
   ¨¹¹ Pointeur:= new Ada.Streams.Stream_Element_Array
```

```
 §                 (1..Ada.Streams.Stream_Element_Offset(Taille));
 ¨¹¹ Ada.Streams.Stream_Io.Create(Flot_Out,
 §                                Ada.Streams.Stream_Io.Out_File,
 §                                "loco2.jpg");
 ¨¹¹ Ada.Streams.Stream_Io.Read(Flot_In,
 §                              Pointeur.all,Ada.Streams.
 §                              Stream_Element_Offset(Taille));
 ¨¹¹ Ada.Streams.Stream_Io.Write(Flot_Out,Pointeur.all);
 ¨¹¹ Ada.Streams.Stream_Io.Close(Flot_In);
 ¨¹¹ Ada.Streams.Stream_Io.Close(Flot_Out);
 ©end Exemple_14;

  with Ada.Text_Io;
  with Ada.Streams.Stream_Io;
 ¬¹11111111
Þßàprocedure Exemple_15 is
 ªË¹1111111
 §
 § ï type T_Promotion is (L1, L2, L3, L4, L5);
 § ï type T_Nom is new String (1 .. 30);
 § ï type T_Prenom is new String (1 .. 10);
 § ï type T_Eleve is
 §      record
 §         Nom       : T_Nom;
 §         Prenom    : T_Prenom;
 §         Promotion : T_Promotion;
 §      end record;
 §
 § ¡¥¥¥¥¥¥¥¥
 §ØÓìpackage Promotion_Es is new Ada.Text_Io.Enumeration_Io
 § ¢    (T_Promotion);
 § £¦¦¦¦¦¦¦¦¦
 §
 § í Flot_In         : Ada.Streams.Stream_Io.File_Type;
 § í Flot_Out        : Ada.Streams.Stream_Io.File_Type;
 § í Ptr_De_Flot_In  : Ada.Streams.Stream_Io.Stream_Access;
 § í Ptr_De_Flot_Out : Ada.Streams.Stream_Io.Stream_Access;
 § í Eleve_Out       : T_Eleve;
 § í Nom_Out         : T_Nom       := (others => ' ');
 § í Prenom_Out      : T_Prenom    := (others => ' ');
 § í Promotion_Out   : T_Promotion := L1;
 § í Nom_In          : T_Nom;
 § í Prenom_In       : T_Prenom;
 § í Promotion_In    : T_Promotion;
 § í Eleve_In        : T_Eleve;
 §begin
 ¨¹¹ Nom_Out(1..8):="Lovelace";
 ¨¹¹ Prenom_Out(1..3):="Ada";
 ¨¹¹ Eleve_Out.Nom:=Nom_Out;
 ¨¹¹ Eleve_Out.Prenom:=Prenom_Out;
 ¨¹¹ Eleve_Out.Promotion:=Promotion_Out;
 ¨¹¹ Ada.Streams.Stream_Io.Create(Flot_Out,
 §                                Ada.Streams.Stream_Io.Out_File,
 §                                "exemple_15.dat");
 ¨¹¹ Ptr_De_Flot_Out:=Ada.Streams.Stream_Io.Stream(Flot_Out);
 ¨¹¹ T_Promotion'Write(Ptr_De_Flot_Out,Promotion_Out);
 ¨¹¹ T_Nom'Write(Ptr_De_Flot_Out,Nom_Out);
 ¨¹¹ T_Prenom'Write(Ptr_De_Flot_Out,Prenom_Out);
 ¨¹¹ T_Eleve'Write(Ptr_De_Flot_Out,Eleve_Out);
 ¨¹¹ Ada.Streams.Stream_Io.Close(Flot_Out);
 ¨¹¹ Ada.Streams.Stream_Io.Open(Flot_In,
 §                              Ada.Streams.Stream_Io.In_File,
```

```
§                               "exemple_15.dat");
¨¹¹ Ptr_De_Flot_In:=Ada.Streams.Stream_Io.Stream(Flot_In);
¨¹¹ T_Promotion'Read(Ptr_De_Flot_In,Promotion_In);
¨¹¹ T_Nom'Read(Ptr_De_Flot_In,Nom_In);
¨¹¹ T_Prenom'Read(Ptr_De_Flot_In,Prenom_In);
¨¹¹ T_Eleve'Read(Ptr_De_Flot_In,Eleve_In);
¨¹¹ Ada.Streams.Stream_Io.Close(Flot_In);
¨¹¹ Ada.Text_Io.Put("Promotion_In : ");
¨¹¹ Promotion_Es.Put(Promotion_In);
¨¹¹ Ada.Text_Io.New_Line;
¨¹¹ Ada.Text_Io.Put_Line("Nom : "&String(Nom_In));
¨¹¹ Ada.Text_Io.Put_Line("Prenom : "&String(Prenom_In));
¨¹¹ Ada.Text_Io.Put("Eleve_In.Promotion : ");
¨¹¹ Promotion_Es.Put(Eleve_In.Promotion);
¨¹¹ Ada.Text_Io.New_Line;
¨¹¹ Ada.Text_Io.Put_Line("Eleve_In.Nom : "&String(Eleve_In.Nom));
¨¹¹ Ada.Text_Io.Put_Line("Eleve_In.Prenom : "&
§                         String(Eleve_In.Prenom));
©end Exemple_15;
```