ISIM-DIG **1ère année**

enseignant : Ph. Reitz
année scolaire : 1995-1996

Le langage Ada

une introduction

LIRMM
161, rue Ada
34392 Montpellier cedex 5

Tél: 67 41 85 85
Fax : 67 41 85 00
E-mail : reitz@lirmm.fr

Introduction

◇ La *crise* du logiciel

Application de grande taille et complexe → processus de conception impérativement **rigoureux**.

▷ Symptômes

– l'application ne répond pas au cahier des charges

– des facteurs économiques clés sont sous-estimés :

  – les coûts dépassent les prévisions
  – les délais ne sont pas tenus

– le logiciel est :

  – difficilement maintenable
  – peu ou pas portable
  – inefficace (consommation en temps ou en mémoire).

▷ Afin d'y remédier : le génie logiciel

– gestion de projets :

  – découpage modulaire
  – gestion de l'historique des modifications
  – auto-documentation

– meilleur qualité des programmes :

  – aide à la spécification, algorithmique
  – langages à haut degré d'abstraction
  – techniques de validation : preuves, tests critiques.

Ada en quelques mots ▷

◇ Historique

Né à la fin des années 70, suite à un appel d'offre en 1974 du DoD (*Department of Defense*) américain → normalisation en 1983 (Ada83).
Premiers compilateurs validés au milieu des années 1980.

◇ Son objectif

Satisfaire aux exigences du monde des logiciels suivantes :

– fiabilité, sécurité

– maintenabilité, qualité

– réutilisabilité

◇ Caractéristiques générales

- fortement typé
- modulaire (→ compilation séparée)
- généricité
- traitement des exceptions
- gestion de la concurrence de tâches
- syntaxe inspirée de PASCAL → lisibilité

◁ Ada en quelques mots

◇ Ada aujourd'hui

– Intégration du paradigme objet → norme ADA95

– Langages concurrents : C++, EIFFEL

– Ses principaux usagers : les grandes institutions (privées ou publiques) de développement et/ou de la recherche appliquée.

◇ Les langages de programmation

Tout un monde : plusieurs centaines de langages existent...

Langages impératifs : ADA, PASCAL, C, FORTRAN, BASIC

Langages fonctionnels : LISP, ML, SCHEME

Langages logiques : PROLOG

Langages à objets : SMALLTALK, C++, EIFFEL

et les autres : les langages à piles (FORTH, POP), d'acteurs (PLASMA, ACTOR), parallèles (OCCAM), à règles (OPS5), ...

Contenu de ce cours

◇ Norme Ada83

Tous les aspects de la norme ADA83 sont présentés, exceptés :

– les spécifications de représentation et de renommage

– toutes les spécificités de l'environnement Alsys-ADA

◇ Norme Ada95

Absolument pas abordée.

◇ Bibliographie (non exhaustive)

[**Dod80**] Ministère Américain de la Défense : *manuel de référence du langage de programmation* ADA. Trad. A. KRUCHTEN & Ph. KRUCHTEN. Éd. Eyrolles, 1982.

[**Barnes88**] J. BARNES : *programmer en* ADA. InterÉditions, 1988.

En langue anglaise, plus d'une cinquantaine d'ouvrages !

Pour ceux qui ont accès à `Internet` :

– `http://lglwww.epfl.ch/Ada/`

  – manuel de référence ADA83 sur `./LRM/83/RM/rm83html/index.html`

  – manuel de référence ADA95 sur `./LRM/9X/Rationale/rat95html/`

– `http://www.acm.org/sigada/`

Présentation générale

◇ Les unités

Concept central d'ADA : l'**unité**.
Construire un programme = construire et assembler des unités

Trois catégories d'unités :

- les **sous-programmes** $\Longrightarrow$ code exécutable
  → les **procédures**
  → les **fonctions**
- les **paquetages** $\Longrightarrow$ composants logiciels
- les **tâches** $\Longrightarrow$ processus concurrents

◇ Élaboration d'un programme en trois phases :

1. **édition** (*edit*) :
   – produire des **fichiers sources** décrivant les unités et leurs agencements
2. **compilation** (*compile*) :
   – produire des **unités compilées**, rangées dans des **librairies**
   – à partir de fichiers sources et de librairies déjà existantes
3. **liaison** (*bind*) :
   – produire le code exécutable du programme
   – à partir de librairies existantes

Présentation générale

◇ Remarques sur les librairies

- Les librairies peuvent être organisées en **familles**.
  → famille = projet dans le monde C ou C++.
- Une librairie est autonome (peut-être exploitée sans le texte source de ses unités)
  → réutilisation - diffusion - commercialisation d'unités (notion de **composant logiciel**)

◇ Le texte source d'une unité

Ce texte source concerne trois aspects d'une unité :

- sa définition
- ses déclarations d'exploitation
- son exploitation effective

Notions générales sur les unités

◇ Une unité :

- porte un **nom** (ou identificateur), satisfaisant la contrainte suivante :

  le nom commence par une lettre, suivie éventuellement de n'importe quel nombre de lettres, de chiffres ou de caractères _ ; le nom ne peut néanmoins pas contenir deux _ consécutifs. Pas de distinction entre majuscules et minuscules.

  Exemple :

noms valides :	***noms invalides :***
`Un_nom`	`1_nom`
`pi`	`_pi`
`Un_petit_texte_pour_nom`	`Un__texte`
`log234_x_28`	

- fait éventuellement l'objet de **déclarations** :

```
partie_déclarative;
```

→ permet de l'identifier et de la caractériser sans ambiguïté afin de pouvoir l'exploiter (compilation / liaison).

→ toute référence à une unité dans un texte source implique nécessairement qu'elle ait été déclarée au préalable.

Exemple :

déclaration d'une fonction :

```
function addition(x, y : integer) return integer;
```

Désormais le texte source peut faire référence à `addition` → *l'unité a parfaitement été identifiée.*

Notions générales

- possède une seule **définition** :

  Une définition est une déclaration complétée de l'implantation de l'unité, i.e. *comment* elle est réalisée.

```
partie déclarative is
        déclarations des entités
begin
        définition du corps
exception
        traitement des exceptions
end nom de l'entité;
```

  → permet de préciser :
  - les entités qu'elle manipule
  - comment elle les manipule dans les cas normaux (corps)
  - son comportement en cas de problèmes (exceptions)

  La partie *définition du corps* est obligatoire.

  Les parties *déclarations des entités* et *traitement des exception* sont optionnelles.

  Exemple :

  définition d'une fonction :

```
function addition(x, y : integer) return integer is
    s : integer;
begin
    s := x+y;
    return s;
end addition;
```


sur les unités

◇ Découpage traditionnel

- déclaration = description de l'interface de l'unité, i.e. comment faire pour l'exploiter
- définition = implantation de l'unité, i.e. comment elle est réalisée

◇ Les entités d'une unité

Une **entité** peut être :

→ un **objet**
- une **constante**
- une **variable**

→ un **type**

→ une **exception**

→ une **sous-unité** (ou unité locale)

Les objets d'une unité

◇ Considérations générales

Une définition d'objet apparaît dans la partie des déclarations d'entités d'une unité ou d'un bloc.

variable / constante : tout objet possède :
- un **nom**
- un **type**
- une **valeur**

Une fois définis pour un objet :
- son nom et son type ne peuvent plus être modifiés
- sa valeur peut être modifiée (objet **variable**) ou pas (objet **constante**).

portée d'un objet : partie du programme pour laquelle la définition d'un objet a un sens.

visibilité d'un objet : un objet, bien que référencé dans sa portée, peut ne pas être visible → par exemple, un autre objet ayant un nom identique a été défini (problème de la **surcharge**).

durée de vie d'un objet : c'est la période durant laquelle un espace mémoire lui est réservé pour coder sa valeur.

→ égale la durée d'activation de l'unité ou bloc dans lequel il a été défini.

Exemple :

au moment opportun...

Les objets d'une unité

◇ Définition d'une constante

```
liste de noms : constant type := expression;
```

type peut être omis ; dans ce cas, chaque nom est un synonyme du littéral associé à la valeur de *expression*.

Exemple :

```
pi               : constant float := 3.1415927;
un, one, ein, uno : constant       := 1;
```

`pi` *est une constante de type* `float` *approximant* π.

`un`, `one`, `ein` *et* `uno` *sont tous synonymes du littéral entier* `1` → *leur type est de la catégorie des types entiers, pas nécessairement du type entier prédéfini* `integer`.

◇ Définition d'une variable

```
liste de noms : type;    -- variables non initialisées

liste de noms : type := expression;
```

Exemple :

```
a, b    : integer;
x, y, z : float := pi;
```

Les types

◇ Introduction

type : décrit un ensemble de **valeurs** manipulables par les unités :

– chaque type possède un système de codage de ses valeurs

– toute opération caractérise les valeurs qu'elle manipule par leur type.

Toute valeur est désignable par un **littéral** et n'a qu'un seul type.
Un littéral peut désigner plusieurs valeurs (→ **surcharge**).

▷ Le monde des types

Deux catégories de types :

– types primitifs (appelés *scalaires* en ADA)

– types construits à partir de types existants

→ spécialisation de type (**sous-type**)

→ clonage de type (**dérivation**)

→ **construction** de type (tableaux, enregistrements, etc).

◇ Conversion - qualification

Soient T un nom de type et e une expression, alors :

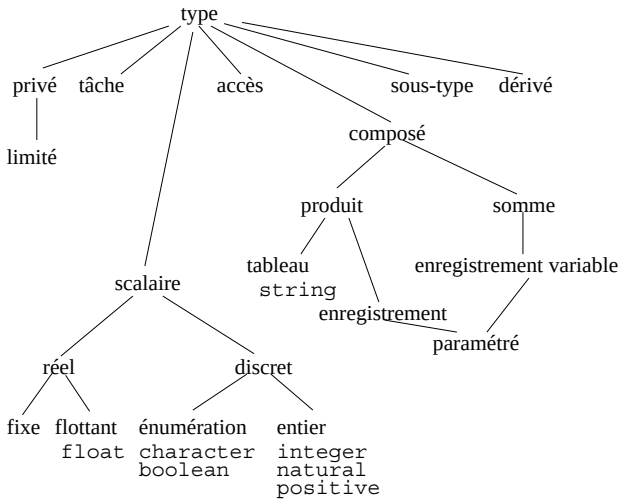
- $T(e)$ convertit la valeur de e en une valeur équivalente de type T (→ **conversion**)
- $T'(e)$ précise que le type de e est T (→ **qualification**)

Les types

◇ Attributs d'un type

Selon le type défini, le compilateur génère automatiquement quelques fonctions utiles donnant des informations sur le type.
→ notion d'**attribut**

◇ Une classification des types



Les sous-types

Définir un **sous-type** d'un type existant (appelé alors **type de base**), c'est :

restreindre l'ensemble des valeurs possibles du type de base.

▷ Forme générale d'une définition

subtype *nom* **is** *type* *contrainte*;

La *contrainte* est optionnelle, et dépend de la nature du type de base.

▷ Propriétés des sous-types

– tout sous-type hérite des littéraux du type de base

– toute valeur d'un sous-type est aussi une valeur du type de base (conversion implicite du sous-type vers le type de base)

→ tout sous-type hérite des unités exploitant son type de base

– les attributs du type de base sont aussi ceux du sous-type

▷ Attribut des sous-types

Si `T` est un sous-type, alors `T'base` désigne son type de base.

Les types dérivés

Dériver un type à partir d'un type existant (appelé le **type parent**), c'est :

définir un nouveau type par simple recopie de toutes les définitions du type parent.

▷ Forme générale d'une définition

type *nom* **is new** *type* *contrainte*;

La *contrainte* est optionnelle, et dépend de la nature du type de base.

▷ Propriétés des types dérivés

– les littéraux du type dérivé sont ceux du type parent

– l'ensemble des valeurs du type dérivé est :

– disjoint de l'ensemble des valeurs du type parent

– en bijection avec ce même ensemble (les 2 types partagent le même codage) si pas de *contrainte*.

Toute valeur du type dérivé peut-être convertie en une valeur du type parent, et inversement ; toutefois, cette conversion doit être *explicitement* écrite.

– toute unité définie sur le type parent est définie à l'identique pour le type dérivé.

– idem pour les attributs du type parent.

Les types scalaires

◇ Caractéristiques

– valeurs d'un type scalaire totalement ordonnées :

→ opérateurs de comparaison : `= /= < <= > >=`

→ il existe une borne inférieure et une borne supérieure.

– si l'évaluation d'une expression d'un type scalaire est une valeur hors des bornes du type, l'exception `constraint_error` est levée.

◇ Attributs des types scalaires

si v est une expression d'un type scalaire T et c une chaîne de caractères, alors :

T`'first`	= plus petite valeur de T
T`'last`	= plus grande valeur de T
T`'image(`v`)`	= chaîne de caractères représentant v
T`'value(`c`)`	= valeur dont la représentation est c

◇ Deux sortes de types scalaires

– les **types discrets**

– les **types réels**

Les types discrets

◇ Attributs des types discrets

Si v est une expression d'un type discret T, et e une expression de type entier, alors :

T`'width`	= nombre de valeurs du type T
T`'pos(`v`)`	= rang (position) de v dans le type
T`'val(`e`)`	= valeur de type T de rang e
T`'succ(`v`)`	= valeur suivant v dans le type
T`'pred(`v`)`	= valeur précédant v dans le type

Tout problème de calcul entraîne la levée de l'exception `constraint_error`.

◇ Notion d'intervalle

Soient `a` et `b` deux expressions d'un même type discret ; un **intervalle** sur ce type s'écrit :

`a .. b` les bornes sont incluses

◇ Deux sortes de types discrets

– les **types énumérés** (ou énumérations)

– les **entiers**

Les types énumérés

◇ Les littéraux

caractéristique d'un type énuméré : sa définition contient *en extension* l'ensemble ordonné de ses littéraux.

Un ***littéral d'énumération*** est soit :

– un nom (identificateur)

– un **caractère littéral** → encadré de `'`

◇ Forme générale d'une définition

type *nom* **is** (*liste de littéraux d'énumération*);

Exemple :

Il existe au moins deux types énumérés prédéfinis en ADA *:*

```
type character is (nul, ..., 'A', 'B', ..., '~');
     -- pseudo-ADA : 128 littéraux explicitement définis
type boolean is (false, true);
```

Quelques exemples de types énumérés :

```
type alphabet is ('a', 'b', 'c', 'd', 'e', 'f', 'g',
                  'h', 'i', 'j', 'k', 'l', 'm', 'n',
                  'o', 'p', 'q', 'r', 's', 't', 'u',
                  'v', 'w', 'x', 'y', 'z');
type voyelle is ('a', 'e', 'i', 'o', 'u', 'y');

type couleur is (rouge, vert, bleu);

type binaire_incertain is ('0', '1', indefini);
```


Les types énumérés

◇ Remarques

– l'ordre sur les littéraux est celui de leur définition

– le rang du premier littéral est `0`

– possibilité de contrôler le codage des littéraux (il doit alors respecter la relation d'ordre) :

Exemple :

```
type couleur is (rouge, vert, bleu);
for couleur use (vert => 2, rouge => 1, bleu => 4);
```

– les attributs `'succ`, `'pred`, etc, sont indépendants du codage.

Exemple :

supposons que soient définis :

```
type couleur is (rouge, vert, bleu);
for couleur use (vert => 2, rouge => 1, bleu => 4);
```

alors :

l'expression ...	*... a pour valeur*
`couleur'first`	`rouge`
`couleur'last`	`bleu`
`couleur'succ(rouge)`	`vert`
`couleur'pos(bleu)`	`2`
`couleur'val(2)`	`bleu`
`couleur'image(rouge)`	*"ROUGE"*
`couleur'value("rouge")`	*rouge*

Les types énumérés

◇ Surcharge des littéraux d'énumération

Tout littéral d'énumération peut être **surchargé** → un même identificateur est exploité dans des définitions de types énumérés distincts.

Exemple :

```
type couleur is (rouge, vert, bleu, orange);
type fruit is (banane, orange, pomme, poire);
```

– *contexte suffisant pour lever l'ambiguïté ⇒ aucun problème*

```
une_couleur : constant couleur := orange;
```

– *contexte insuffisant ⇒* ***qualifier*** *l'expression :*

```
un_fruit : constant := fruit'(orange);
```

◇ Sous-type d'un type énuméré

Définir un sous-type d'un type énuméré, c'est en restreindre l'ensemble des valeurs définies en en spécifiant un sous-intervalle.

subtype *nom* **is** *type* **range** *expr.* .. *expr.*

Exemple :

```
type jour is (lun, mar, mer, jeu, ven, sam, dim);

subtype jour_ouvrable is jour range lun..ven;
-- impossible d'ecrire :
--  subtype jour_plein is jour_ouvrable (lun..mer, ven);
--  subtype jour_plein is jour_ouvrable (lun, mar, mer, ven);

subtype lettre is character range 'a'..'z';
-- impossible : subtype voyelle is lettre ('a', 'e', 'i', 'o', 'u', 'y'
```


Les types énumérés

◇ Opérateurs prédéfinis sur le type `boolean`

▷ Pour le type `boolean` seul

Soient `a` et `b` deux expressions de type `boolean` :

not `a`	négation logique
`a` **and** `b`	et logique complet
`a` **and then** `b`	et logique partiel (évaluation au besoin)
`a` **or** `b`	ou logique complet
`a` **or else** `b`	ou logique partiel (évaluation au besoin)
`a` **xor** `b`	ou exclusif

▷ Pour tout type discret

Soient `a` une expression d'un type discret T et `b` un intervalle sur T :

`a` **in** `b`	test d'appartenance
`a` **not in** `b`	test de non appartenance

▷ Pour tout type scalaire

Soient `a` et `b` deux expressions d'un même type scalaire :

`a > b`	`true` si `a` strictement supérieur à `b`
`a >= b`	idem avec supérieur ou égal
`a < b`	idem avec strictement inférieur
`a <= b`	idem avec inférieur ou égal

▷ Pour tout type

Soient `a` et `b` deux expressions d'un type T quelconque :

`a = b`	prédicat d'égalité
`a /= b`	prédicat d'inégalité

Les types énumérés

Exemple :

L'expression ...	*... a pour valeur:*
not `true`	`false`
`true` **and** `false`	`false`
`false` **and** `true`	`false`
`false` **and** `(1/0 = 1)`	*erreur (exception* `numeric_error`*)*
`true` **and then** `false`	`false`
`false` **and then** `(1/0 = 1)`	`false`
`false` **or** `true`	`true`
`true` **or** `(1/0 = 1)`	*erreur (exception* `numeric_error`*)*
`true` **or else** `false`	`true`
`true` **or else** `(1/0 = 1)`	`true`
`true` **xor** `true`	`false`
`true` **xor** `false`	`true`
`3` **in** `0..10`	`true`
`2` **not in** `-10..10`	`false`

Les types entiers

◇ Les littéraux entiers

Un littéral entier s'écrit classiquement comme une séquence de chiffres, sachant que :

– le caractère '_' peut être inséré entre 2 chiffres consécutifs

– un exposant peut être précisé → lettre `E` (ou `e`) suivie d'un entier naturel

– possibilité d'exploiter une base de numération autre que 10, comprise entre 2 et 16 (inclus)

Exemple :

Petit panorama des littéraux entiers par l'exemple :

```
10000     10_000          1E4
   31     2#0001_1111#    16#1F#
  224     2#111#E5        16#E#E1    8#340#
```

◇ Forme générale d'une définition de type

type *nom* **is range** *intervalle*;

Exemple :

Un type entier prédéfini en ADA *:*

```
type integer is range -32768..32767;
```

Les bornes indiquées sont dépendantes des systèmes. Elles sont facilement connaissables (`integer'first` *et* `integer'last`*).*

Les types entiers

◇ Sous-type d'un type entier

Définir un sous-type d'un type entier, c'est en restreindre les bornes.

subtype *nom* **is** *type* **range** *expression*..*expression*;

Exemple :

Les deux types suivants sont prédéfinis en ADA *:*

```
subtype natural is integer range 0..integer'last;
subtype positive is natural range 1..natural'last;
```

◇ Opérateurs prédéfinis

Soient `a` et `b` deux expressions de type entier :

`+ a`	identité
`- a`	opposé
`a + b`	addition
`a - b`	soustraction
`a * b`	multiplication
`a / b`	quotient de la division entière
`a rem b`	reste de la division entière
`a mod b`	modulo
`a ** b`	exponentiation (`b : natural`)
`abs(a)`	valeur absolue

Tout problème de calcul entraîne la levée de l'exception `numeric_error`.

Les types réels

◇ Les littéraux réels

Littéral réel : de la forme *mantisse exposant*

- *mantisse* : de la forme *entier* . *entier* → partie entière puis partie fractionnaire, séparées par '.'
- *exposant* : optionnel, de la forme `E` *entier signé* → désigne la puissance de 10 (si base décimale) devant être multipliée à la mantisse.

Exemple :

```
5.0          0.5E1          0.5E+1          50.0E-1
3.141_592_7
255.0        16#F.F#E+1     2#1111_11110.0#E-1
```

◇ Opérateurs prédéfinis

`+ a`	identité
`- a`	opposé
`a + b`	addition
`a - b`	soustraction
`a * b`	multiplication
`a / b`	division
`a ** b`	exponentiation (`b : integer`)
`abs(a)`	valeur absolue

Tout problème de calcul entraîne la levée de l'exception `numeric_error`.

◇ Deux sortes de types réels

- les **types à virgule flottante** (plus simplement **flottants**)
- les **types à virgule fixe**

Les types réels flottants

◇ Forme générale d'une définition

type *nom* **is digits** *expr*;
type *nom* **is digits** *expr* **range** *expr*..*expr*;

après **digits** : nombre de chiffres de la mantisse.
après **range** : bornes.

◇ Attributs des types flottants

Si T est un type flottant, alors :

T`'digits`	= nombre de chiffres de la mantisse
T`'mantissa`	= nombre de bits codant la mantisse
T`'emax`	= valeur de l'exposant max
T`'small`	= plus petite valeur strictement positive
T`'large`	= plus grande valeur positive
T`'epsilon`	= 1$^{\text{ière}}$ valeur supérieure à T`'small`

Exemple :

Un type flottant prédéfini en ADA *(dépendant des systèmes) :*

```
type float is digits 7 range -1e37..1e37;
```

◇ Sous-type d'un type flottant

Définir un sous-type d'un type réel flottant, c'est en restreindre les bornes.

Exemple :

```
subtype positive_float is float range 0..float'last;
```


Les types réels fixes

◇ Forme générale d'une définition

type *nom* **is delta** *expr*;
type *nom* **is delta** *expr* **range** *expr*..*expr*;

après **delta** : écart entre deux valeurs successives.
après **range** : bornes.

◇ Attributs des types réels fixes

Si T est un type réel fixe, alors :

T`'delta`	= écart entre deux valeurs successives
T`'bits`	= nombre de bits nécessaires au codage
T`'large`	= plus grande valeur positive

Exemple :

```
type pression is delta 0.001 range 0.0 .. 10.0;
```

◇ Sous-type d'un type réel fixe

Définir un sous-type d'un type réel fixe, c'est en restreindre les bornes.

Exemple :

```
subtype haute_pression is pression
                      range 5.0..pression'last;
```

Les instructions

◇ Composition des corps d'unités

Le **corps** d'une unité est composé d'une séquence d'**instructions**.

◇ Les instructions

Classées en six catégories :

- l'**affectation**
- les **conditionnelles** (2 formes)
- les **boucles** (3 formes)
- les **blocs**
- l'**appel** aux procédures
- les **instructions spéciales**

Toute instruction peut être précédée d'une **étiquette** de la forme <<*nom*>>

Les instructions

◇ Qu'est-ce qu'*exécuter* une instruction

étant donnés :

- une liste d'objets en mémoire
- une instruction à exécuter

alors :

- transformer la mémoire selon la nature de l'instruction en cours :
  - changer les valeurs des objets variables existants
  - ajouter ou enlever des objets

Décrire les effets de l'exécution d'une instruction, c'est en donner sa **sémantique**.

◇ Qu'est-ce qu'exécuter une séquence d'instructions

étant donnés :

- une liste d'objets en mémoire
- une séquence d'instructions à exécuter

alors :

- exécuter la première instruction de la séquence
- exécuter le reste de la séquence.
  Chaque instruction travaille sur la mémoire modifiée par l'exécution de l'instruction précédente.

L'affectation

◇ Forme générale

```
nom := expression;
```

- *nom* désigne le nom d'un objet variable.
- La valeur de *expression* doit appartenir à celles définies par le type de l'objet.

◇ Sémantique

1. évaluation de *expression* → valeur v
2. contrôle de l'appartenance de v à l'ensemble des valeurs définies pour le type de la variable → levée de l'exception `constraint_error` si échec.
3. modification de la variable *nom* : sa valeur est désormais v.

Exemple :

Supposons définis :

```
a : integer        := -1;
b : natural        := 2;
c : constant float := 0.0;
```

Les affectations suivantes :

```
a := b;   -- correcte
b := a;   -- incorrecte : pas de conversion implicite type vers sous-type
b := natural(a); -- correcte, mais lève constraint_error à l'exécution
c := 1.0; -- incorrecte : c est une constante
a := c;   -- incorrecte : types incompatibles
a := integer(c); -- correcte
```


La conditionnelle - $1^{ère}$ forme

◇ Forme générale

```
if condition 0 then
    séquence d'instructions 0
elsif condition 1 then
    séquence d'instructions 1
elsif condition 2 then
    ...
elsif condition n then
    séquence d'instructions n
else
    séquence d'instructions n+1
end if;
```

Les parties **elsif** et **else** sont optionnelles. Toutes les *conditions* sont des expressions de type `boolean`.

◇ Sémantique

Évalue en séquence chacune des *conditions* ; soit i la condition en cours d'examen (au début, $i = 0$) ; examinons la valeur de *condition i* :

- si cette valeur est `true`, alors exécute la *séquence d'instructions i* → l'exécution de la conditionnelle **if** est alors achevée.
- si cette valeur est `false`, alors :
  - si $i = n$, alors exécute la *séquence d'instructions n+1* → l'exécution de la conditionnelle **if** est alors achevée.
  - sinon (donc $i < n$) ré-appliquer ce plan d'action avec $i + 1$

La conditionnelle

Exemple :

supposons que soient définies les entités suivantes (jeu de belote) :

```
type une_Figure is (sept, huit, neuf, dix, valet, dame, roi, as);
type une_Couleur is (pique, carreau, coeur, trefle);
subtype des_Points_Carte is natural range 0 .. 20;

couleur_carte, couleur_atout : une_Couleur;
figure_carte                 : une_Figure;
points_carte                 : des_Points_Carte;
```

*L'instruction suivante permet de compter les points associés à une carte donnée (*couleur_carte *et* figure_carte*) :*

```
if figure_carte in sept..huit then
   points_carte := 0;
elsif figure_carte = neuf then
   if couleur_carte = couleur_atout
       then points_carte := 14;
       else points_carte := 0;
   end if;
elsif figure_carte = dix then
   points_carte := 10;
elsif figure_carte = valet then
   if couleur_carte = couleur_atout
       then points_carte := 20;
       else points_carte := 2;
   end if;
elsif figure_carte = dame then
   points_carte := 3;
elsif figure_carte = roi then
   points_carte := 4;
else -- ici figure_carte = as, obligatoirement
   points_carte := 11;
end if;
```

Pour ce problème particulier, c'est loin d'être la meilleure solution !

La conditionnelle - $2^{ème}$ forme

◇ Forme générale

```
case expression is
      when liste d'expr. 1 => séq. d'instructions 1
      ...
      when liste d'expr. n => séq. d'instructions n
      when others => séquence d'instructions n+1
end case;
```

La partie **others** est optionnelle si toutes les alternatives ont été considérées.

Une liste d'expressions est soit :

– une expression seule

– un intervalle

– une séquence d'expressions séparées par |.

Toute expression figurant après un **when** doit pouvoir être évaluée au moment de la compilation → aucun appel de fonction → le compilateur vérifie que tous les cas sont traités.

◇ Sémantique

1. évaluer *expression* → valeur v
2. repérer la clause **when** traitant la valeur v (soit i son rang) ; exécuter alors la *séq. d'instructions i* ; l'exécution de la conditionnelle **case** est alors achevée.

La conditionnelle

Exemple :

Reprenons l'exemple précédent du jeu de belote ; l'instruction suivante calcule le même résultat (calcul des points associés à une carte) :

```
case figure_carte is
     when sept | huit => points_carte := 0;
     when neuf        =>
                 if couleur_carte = couleur_atout
                     then points_carte := 14;
                     else points_carte := 0;
                 end if;
     when dix         => points_carte := 10;
     when valet       =>
                 if couleur_carte = couleur_atout
                     then points_carte := 20;
                     else points_carte := 2;
                 end if;
     when dame        => points_carte := 3;
     when roi         => points_carte := 4;
     when as          => points_carte := 11;
end case;
```

Cette solution est encore loin d'être la plus courte !

Le bloc

◇ Forme générale

```
nom :
declare
      déclarations d'entités
begin
      séquence d'instructions
exception
      traitement des exceptions
end nom;
```

Sont optionnels : le nom du bloc, les déclarations d'entités, le traitement des exceptions.

◇ Sémantique

1. si des *déclarations d'entités* sont présentes, **élaboration** de toutes les entités qui y sont déclarées.
2. exécution de la *séquence d'instructions* ; si une exception est levée, soit :
   – elle est traitée dans *traitement des exceptions*, et l'exécution du bloc se termine normalement
   – soit elle n'est pas traitée ; l'exécution se termine en levant la même exception dans le bloc appelant.
3. destruction des entités élaborées en 1.

Le bloc

◇ Portée/visibilité des déclarations/définitions

– portée des entités déclarées : comprend le corps et le traitement des exceptions du bloc.

– si un nom donné à une entité référençait déjà une autre entité dans une unité ou bloc englobant, la nouvelle définition masque l'ancienne dans la portée (l'ancienne n'est plus visible, à moins d'indiquer son **chemin de nommage**).

Exemple :

```
B1x : declare
    a  : integer := 1;
begin
    B11 : declare
        b : integer := 2;
    begin
        corps de B11 -- a visible, ainsi que b
    exception
        traitements  -- a et b toujours visibles
    end B11;
    -- b n'est plus dans la portée, a l'est encore
    B12 : declare
        a : character := 'x';       -- B1x.a masquée
        b : character := a;         -- il s'agit de B1x.B12.a
        -- B11.b hors de portée
        c : float := float(B1x.a); -- B1x.a visible tout de même !
    begin
        corps de B12
    end B12;
    suite du corps de B1x -- seule variable à portée : B1x.a
end B1x;
```


Sur la notion d'élaboration

La notion d'**élaboration** n'a de sens que lors de l'exécution du programme.

Élaborer une entité déclarée dans un bloc ou une unité, c'est, selon que l'entité est :

un objet : calcul de l'expression initialisant l'objet, puis réservation en mémoire (allocation) de l'espace nécessaire au codage de la valeur associée à cet objet

un type : ne rien faire

une exception : ne rien faire

une sous-unité : dépend de la nature de l'unité

La boucle

◇ Forme générale

```
nom :
itérateur loop
      séquence d'instructions
end loop nom;
```

Le nom de la boucle est optionnel.

La séquence d'instructions peut contenir l'instruction spéciale **exit** sous l'une des formes suivantes :

– **exit** *nom*;

– **exit** *nom* **when** *condition*;

→ équivalent à **if** *condition* **then exit** *nom*; **endif**;

▷ Sémantique de loop sans itérateur

```
nom : loop
      séquence d'instructions
end loop nom;
```

1. exécute en séquence chaque instruction de la boucle :
   – Si l'instruction exécutée est **exit** *nom*, l'exécution de la séquence est interrompue → l'exécution de **loop** est terminée (sortie de la boucle). Idem si la boucle ne portait pas de nom et que l'instruction **exit** seule (sans *nom*) n'était pas elle-même dans une boucle.
2. si aucune instruction **exit** n'a été rencontrée, recommencer en 1.

Une telle boucle peut donc ne jamais s'arrêter.

La boucle

Exemple :

Cette boucle écrit 10 fois Coucou *à l'écran sur une même ligne :*

```
declare
   n : integer := 0;
begin
   loop
      put("Coucou"); -- instruction d'affichage d'un texte
      n := n+1;
      exit when n=10;
   end loop;
end;
```

La boucle suivante écrit 3 lignes, avec 4 Coucou *affichés par ligne :*

```
declare
   l : integer := 0;
begin
   loop
      declare
         n : integer := 0;
      begin
         loop
            put("Coucou");
            n := n+1;
            exit when n=4;
         end loop;
      end;
      new_line; -- instruction forçant l'écran à passer à la ligne suivante
      l := l+1;
      exit when l=3;
   end loop;
end;
```

La boucle

Exemple :

La boucle suivante affiche les entiers naturels par ordre croissant, en commençant par 0 ; les entiers sont affichés à raison de 10 nombres par ligne. L'affichage se termine dès que la somme de tous les entiers affichés est ≥ 1000.

```
declare
    n : natural := 0; -- repère le dernier entier considéré
    s : natural := 0; -- contient la somme de tous les entiers affichés
begin
    affichage : loop
       declare
          c : natural := 0;
              -- nombre d'entiers affichés sur la ligne en cours
       begin
          loop
             exit affichage when s >= 1000;
             put(n); -- instruction d'affichage du contenu d'une variable
             s := s+n;
             n := n+1;
             c := c+1;
             exit when c = 10;
          end loop;
       end;
       new_line; -- force le passage à la ligne suivante
    end loop affichage;
end;
```


La boucle

▷ Itérateur while

```
nom : while condition loop
        séquence d'instructions
      end loop nom;
```

est équivalent à :

```
nom : loop
        exit nom when not condition;
        séquence d'instructions
      end loop nom;
```

▷ Itérateur for ... in

```
nom : for id in intervalle loop
        séquence d'instructions
      end loop nom;
```

est équivalent à (soit `T` le type discret de l'intervalle) :

```
declare
    id : T := T'first;
begin
  if id <= T'last then nom : loop
    séquence d'instructions
    exit nom when id = T'last;
    id := T'succ(id);
  end loop nom; end if;
end;
```

Toute affectation de *Id* dans *séquence d'instructions* est interdite

La boucle

▷ Itérateur for ... in reverse

```
nom : for id in reverse intervalle loop
        séquence d'instructions
      end loop nom;
```

est équivalent à (soit `T` le type discret de l'intervalle) :

```
declare
    id : T := T'last;
begin
  if id >= T'first then nom : loop
    séquence d'instructions
    if id = T'first
        then exit nom;
        else id := T'pred(id);
    end if;
  end loop nom; end if;
end;
```

Toute affectation de *Id* dans *séquence d'instructions* est interdite

◇ Intérêts des itérateurs

– concision → lisibilité

– programmation structurée.

– en général, évite le nommage explicite de boucles.

La boucle

Exemple :

Reprenons les 3 exemples précédents sous forme de boucles avec itérateurs.

– *10 affichages de* Coucou *:*

```
for n in 1..10 loop
   put("Coucou");
end loop;
```

– *affichage de 3 lignes, avec 4* Coucou *par ligne :*

```
for l in 1..3 loop
   for n in 1..4 loop
      put("Coucou");
   end loop;
   new_line;
end loop;
```

– *affichage des entiers jusqu'à somme ≥ 1000 :*

```
declare
    n : natural := 0;
    s : natural := 0;
begin
    affichage : loop
       for c in 1..10 loop
          exit affichage when s >= 1000;
          put(n);
          s := s+n;
          n := n+1;
       end loop;
       new_line;
    end loop affichage;
end;
```

Les instructions spéciales

◇ L'instruction nulle

```
null;
```

Son exécution n'a aucun effet.

◇ Levée d'exception

– levée d'une exception de nom *exception* :

```
raise exception;
```

– relevée d'une exception en cours de traitement :

```
raise;
```

◇ Les instructions liées aux tâches

plus tard...

Les instructions spéciales

◇ Instructions liées aux sous-programmes

▷ Pour les procédures

- appel à une procédure appelée *nom* :

```
nom liste de paramètres;
```

La liste des paramètres est :

– soit vide

– soit de la forme (*expr*) ou (*expr*, ..., *expr*) : autant d'*expr* que de paramètres définis pour *nom*.

- fin de procédure :

```
return;
```

Instruction autorisée seulement dans la définition d'une procédure.

▷ Pour les fonctions

- renvoi du résultat d'une fonction :

```
return expression;
```

Instruction autorisée seulement dans la définition d'une fonction.

Les instructions spéciales

◇ Saut

```
goto étiquette;
```

L'exécution se poursuit à partir de l'instruction possédant l'étiquette indiquée.

Exemple :

```
declare  -- au temps héroïque du BASIC...
     a : integer := 5;
     b : integer := 3;
     s : integer := a;
begin  -- ...la programmation spaghetti
     <<debut>> if b=0 then
        goto fin;
     else
        s := s+1;
        b := b-1;
        goto debut;
     end if;
     <<fin>> null;
end;
```

Proverbe : *point de* **goto** *tu écriras...*

Les sous-programmes

◇ Deux formes

– les **procédures**

– les **fonctions**

◇ Motivations

Abstraire le code : éviter de récrire plusieurs fois un même bloc d'instructions agissant sur des objets différents.

◇ Différences

– une procédure est appelée à un niveau instruction (instruction d'appel)

– une fonction est appelée à un niveau évaluation d'expression → doit impérativement **retourner une valeur**.

◇ Points communs

– règles de passage des paramètres

– valeurs par défaut

– généricité / instanciation

– résolution de la surcharge

– instruction de retour

– ...

Les fonctions

◇ Déclaration

function *nom entête* **return** *type*;

La partie entête spécifie les paramètres de la fonction ; elle est soit vide, soit de la forme :

(*paramètre 1* ; ... ; *paramètre n*)

▷ Forme générale d'un paramètre

liste de noms : *type*
liste de noms : *type* := *expression*

L'expression doit être évaluable à la compilation.

Exemple :

```
function exemple(a, b, c : natural; x, y : float := 1.0)
         return float;
```


Les fonctions

◇ Définition

function *nom entête* **return** *type* **is**
    partie déclarative
begin -- *voir présentation générale des unités page 8*
    corps
exception
    traitement des exceptions
end *nom*;

corps et *traitement des exceptions* doivent contenir au moins une instruction spéciale de retour du résultat, de la forme :

return *expression*;

Exemple :

```
function exemple(a, b, c : natural; x, y : float := 1.0)
         return float is
begin
     if a=b then
         return x;
     elsif b=c then
         return y;
     else
         return x-y;
     end if;
end exemple;
```


Les fonctions

◇ Appel d'une fonction

Supposons une fonction *nom* déclarée ainsi :

function *nom*(*P 1* : *T 1*;
    P 2 : *T 2*;
    ... -- *autres paramètres*
    P k : *T k*;
    P k+1 : *T k+1* := *expression k+1*;
    ... -- *autres paramètres*
    P n : *T n* := *expression n*) **return** *T n+1*;

La fonction possède :

– n paramètres en tout

– k paramètres (de 1 à k) n'ont pas de valeurs par défaut

– les $n - k$ paramètres qui restent en ont

Un appel de fonction figure nécessairement dans une expression de calcul. Trois formes d'appel :

– appel sous forme d'agrégat positionné

– appel sous forme d'agrégat nommé

– appel mixte (positionné & nommé)

Les fonctions

▷ Appel sous forme d'agrégat positionné

Avec la fonction :

function *nom*(*P 1* : *T 1*;
    P 2 : *T 2*;
    ... -- *autres paramètres*
    P k : *T k*;
    P k+1 : *T k+1* := *expression k+1*;
    ... -- *autres paramètres*
    P n : *T n* := *expression n*) **return** *T n+1*;

Un tel appel est de la forme :

nom(*expr 1*, *expr 2*, ..., *expr m*)

avec $k \leq m \leq n$: il y a au moins autant d'*expr*essions qu'il y a de paramètres sans valeurs par défaut, et au plus n.

Les valeurs par défaut complètent les *expr* manquants dans l'appel.

Exemple :

Quelques appels sous forme d'agrégat positionné à la fonction `exemple` *définie précédemment :*

```
3.0+exemple(2, 3, 4)
-- équivalent à 3.0+exemple(2, 3, 4, 1.0, 1.0)

exemple(1, 2, 3, 3.0)
-- équivalent à exemple(1, 2, 3, 3.0, 1.0)

2.0+exemple(0, 1, 1, exemple(1, 2, 3, -1.0, 4.0)-3.0, 1.0)
-- les paramètres sont au complet, pour chaque appel
```

Les fonctions

▷ Appel sous forme d'agrégat nommé

Avec la fonction :

```
function nom(P 1 : T 1;
             P 2 : T 2;
             ... -- autres paramètres
             P k : T k;
             P k+1 : T k+1 := expression k+1;
             ... -- autres paramètres
             P n : T n := expression n) return T n+1;
```

Un tel appel est de la forme :

```
nom(nom 1 => expr 1,
    nom 2 => expr 2,
    ...     -- et ainsi de suite
    nom m => expr m)
```

avec $k \leq m \leq n$; Les valeurs par défaut complètent les *expr* manquants dans l'appel.

Chaque *nom i* est l'un des noms des paramètres de la fonction ; l'ordre n'a pas d'importance.

Exemple :

Quelques appels sous forme d'agrégat nommé à la fonction `exemple` *définie précédemment :*

```
3.0+exemple(b => 3, a => 2, c => 4)
-- équivalent à 3.0+exemple(2, 3, 4, 1.0, 1.0)

exemple(y => 3.0, b => 2, c => 3, a => 1)
-- équivalent à exemple(1, 2, 3, 1.0, 3.0)
```


Les fonctions

▷ Appel mixte

C'est un mélange des 2 formes d'appel précédentes : l'appel

– commence comme un appel à agrégat positionné

– et finit comme un appel à agrégat nommé

Exemple :

Quelques appels sous forme mixte à la fonction `exemple` *définie précédemment :*

```
3.0+exemple(2, c => 4, b => 3)
-- équivalent à 3.0+exemple(2, 3, 4, 1.0, 1.0)

exemple(1, 2, y => 4.0, c => 3)
-- équivalent à exemple(1, 2, 3, 1.0, 4.0)
```


Les fonctions

◇ Sémantique d'un appel de fonction

Supposons une fonction définie par :

```
function nom (P 1 : T 1; ...; P n : T n) return T is
   déclaration d'entités
begin
   corps
exception
   traitement des exceptions
end nom;
```

Comment le résultat d'un appel à la fonction est-il obtenu ?

Supposons que cet appel soit complet (usage des valeurs par défaut si besoin) et qu'il peut s'écrire sous la forme :

```
nom ( expr 1, ..., expr n )
```


Les fonctions

Le résultat de l'appel s'obtient :

1. en calculant chaque *expr i* → valeur obtenue *v i*
2. en exécutant ensuite le bloc suivant :

```
declare
   P 1 : constant T 1 := v 1;
   ...
   P n : constant T n := v n;
   res : T;    -- le résultat
   déclarations des entités
begin
   corps
   <<fin>> null;
exception
   traitement des exceptions
end;
```

sachant que toute instruction **return** *exp* doit être remplacée par :

```
res := exp; goto fin;
```

3. le résultat de l'appel est la valeur de *res*.

◇ Remarques

Les paramètres sont considérés comme des **constantes** → les parties *corps* et *traitement des exceptions* ne peuvent donc pas contenir d'instruction d'affectation sur l'un des *P i*.

Les fonctions

Exemple :

Reprenons notre fonction `exemple` *:*

```
function exemple(a, b, c : natural; x, y : float := 1.0)
        return float is
begin
     if a=b then
         return x;
     elsif b=c then
         return y;
     else
         return x-y;
     end if;
end exemple;
```

En fin d'exécution de ce bloc :

```
declare
    a : integer := 3;
    x : float   := 1.0;
    z : float   := 2.0;
begin
    x := x + exemple(a, a, a, z);
    z := 2.0 * exemple(1, a, a-1, exemple(1, 2, 3, x, x)-1.0, 2.0);
end;
```

nous avons :

Variable	*Valeur*
x	*3.0*
z	*-6.0*

Les procédures

◇ Déclaration

procedure *nom entête*;

La partie entête spécifie les paramètres de la procédure ; elle est soit vide, soit de la forme :

(*paramètre 1* ; ... ; *paramètre n*)

▷ Forme générale d'un paramètre

liste de noms : *mode type*
liste de noms : *mode type* := *expression*

où *mode* est au choix :

- rien : il est considéré alors comme **in**.
- **in** : les noms dénoteront des constantes dont la valeur aura été spécifiée lors de l'appel (soit explicitement, soit par défaut).
- **out** : les noms dénoteront des variables ; seule opération autorisée : l'affectation. Aucun de ces noms ne pourra figurer dans une expression (lecture de la valeur interdite).

  Les paramètres d'appels sont impérativement des variables. Valeur par défaut interdite.
- **in out** : les noms dénoteront des variables ; toute opération autorisée.

  Les paramètres d'appels sont impérativement des variables. Valeur par défaut interdite.

Les procédures

◇ Définition

procedure *nom entête* **is**
    partie déclarative
begin -- *voir présentation générale des unités page 8*
    corps
exception
    traitement des exceptions
end *nom*;

corps et *traitement des exceptions* peuvent contenir des instructions spéciales de retour :

return; -- *sans aucun argument*

Exemple :

```
procedure exemple(a, b : in natural := 1;
                  x, y : in out float;
                  z : out float) is
begin
     if a=b then
         z := x+y;
     elsif b>a then
         z := y;
         return;
     end if;
     y := x;
end exemple;
```


Les procédures

◇ Appel d'une procédure

Un appel de procédure est une instruction.
Syntaxe d'appel similaire à celle des fonctions → agrégat positionné, nommé ou mixte.

Exemple :

```
procedure test (a : integer;
                b : out integer;
                c : in out integer) is
begin
     a := 1;      -- interdit : a est une constante
     b := 1;      -- OK
     c := 1;      -- OK
     c := b;      -- interdit : b non lisible
end test;
```

Quelques appels (en supposant les erreurs corrigées) :

```
declare
    x, y, z : integer := 2;
begin
    -- appel positionné
    test(x, y, z);
        -- OK

    -- appel nommé
    test(b => y-1, a => x+2, c => z);
        -- incorrect : y-1 n'est pas un nom de variable

    -- appel mixte
    test(y+x, c => z+1, b => z);
        -- incorrect : z+1 n'est pas un nom de variable
end;
```

Les procédures

◇ Sémantique d'une instruction d'appel

Supposons une procédure définie par :

```
procedure nom (
          C 1 : in S 1; ...; C k : in S k;
          L 1 : out T 1; ...; L m : out T m;
          V 1 : in out U 1; ...; V n : in out U n )
is
   déclaration d'entités
begin
   corps
exception
   traitement des exceptions
end nom;
```

Un appel à la procédure *nom* s'écrit :

```
nom (expr 1, ..., expr k, X 1, ..., X m, Y 1, ..., Y n);
```

sachant que :

- les *expr i* sont des expressions quelconques (mode **in** des paramètres correspondants)
- les *X i* sont impérativement des noms de variables (mode **out**)
- idem pour les *Y i* (mode **in out**)

Les procédures

L'exécution de l'appel suit le principe suivant :

1. calculer chaque *expr i* → valeur obtenue *v i*
2. noter la valeur de chaque *Y j* → valeur *w j*
3. remplacer l'appel à la procédure par le bloc suivant :

```
declare
   C 1 : constant S 1 := v 1
   ...
   C k : constant S k := v k
   déclarations des entités
begin
   V 1 := w 1;
   ... -- en principe, les V j sont indéfinies ; voir étape 4
   V n := w n;
   corps
   <<fin>> null;
exception
   traitement des exceptions
end;
```

   sachant que toute instruction **return;** doit être remplacée par l'instruction **goto** *fin*; dans *corps* et *traitement des exceptions*

4. dans ce bloc, remplacer :
   - toute occurence de *L i* par *X i*
   - toute occurence de *V i* par *Y i*
5. exécuter le bloc ainsi modifié ; l'appel est terminé.

Les procédures

Exemple :

```
procedure exemple(a : in natural := 1;
                  x : in out float;
                  z : out float) is
begin
    if a=1 then
        z := x;
    elsif a>2 then
        z := x-1.0;
        x := 2.0 * x;
    end if;
end exemple;
```

En fin d'exécution de ce bloc :

```
declare
   a    : integer := 1;
   x, y : float   := 1.0;
begin
   exemple(a+2, z => y, x => x);  -- équivalent à exemple(a+2, x, y);
   exemple(1, y, x);
end;
```

les variables contiennent les valeurs suivantes :

x	*0.0*
y	*0.0*

Surcharge

◇ Règle générale

Tout nom de sous-programme peut être surchargé, i.e. être utilisé pour désigner des sous-programmes dont seuls les entêtes diffèrent.

Exemple :

```
function max (a : integer; b : integer) return integer is
begin
    if a>b then return a; else return b; end if;
end max;

function max (a : float; b : float) return float is
begin
    if a>b then return a; else return b; end if;
end max;
```

◇ Surcharge des opérateurs prédéfinis possible

Exemple :

Les opérateurs ne sont pas réservés aux types prédéfinis; `deux + deux` *vaut* `beaucoup`, *sachant que :*

```
type nombre is (un, deux, beaucoup);

function "+" (a, b : nombre) return nombre is
begin
    if a>b
        then return b+a;
    elsif a=un and then b=un
        then return deux;
        else return beaucoup;
    end if;
end "+";
```

Les types composés

◇ Deux sortes de types composés

- les **types produits**
  - → les **tableaux**
  - → les **enregistrements**
- les **types sommes**
  - → les **enregistrements variables**

◇ Sur la composition de types

En théorie des types, il existe trois façons de combiner des types :

produit ($A \times B$) **:** équivalent du produit cartésien ensembliste
somme ($A + B$) **:** équivalent de l'union disjointe ensembliste
exponentiation ($A \to B$ **ou** B^A) **:** fonctions de A vers B

Les types tableaux

type Tableau = produit de types tous identiques.
valeur Tableau = valeur décomposable en valeurs (**composantes**) toutes de même type.
Accès à une composante via un système d'**index** (ou **coordonnée**).

◇ Forme générale d'une définition

type *nom* **is array** (*liste de dimensions*) **of** *type*;

Une **dimension** est un intervalle sur un type discret ; elle peut prendre l'une des formes suivantes :

type	: nom d'un type discret
intervalle	: type discret contraint
type **range** *intervalle*	: type discret contraint
type **range <>**	: type discret non contraint

Si une dimension d'un tableau est un type discret non contraint, le tableau est dit **non contraint**.

Exemple :

Le type suivant est prédéfini en ADA *:*

```
type string is array (positive range <>) of character;
     -- type tableau non contraint
```

Autres exemples :

```
type RVB is (rouge, vert, bleu);
type mire is array (1..1024, 1..512, RVB)
     of natural range 0..255;
     -- type tableau contraint

type vecteur is array (positive range <>) of float;
     -- type tableau non contraint
```


Les tableaux

◇ Attributs des types tableaux

Si x est le nom d'un type tableau ou d'un objet de type tableau, et d une expression entière non négative, alors :

x`'first(`d`)`	= borne inférieure de la dimension d
x`'last(`d`)`	= borne supérieure de la dimension d
x`'range(`d`)`	= intervalle de la dimension d
x`'length(`d`)`	= nombre d'index de la dimension d

Si (*d*) est omis, alors il est considéré que $d = 1$.

Exemple :

Supposons définies les entités suivantes :

```
type tab1 is array (3..15) of natural;
var1 : tab1;
type tab2 is array (2..11, 5..6) of tab1;
var2 : tab2;
```

Alors :

l'expression...	*...a pour valeur*
`tab1'first(1)`	*3*
`var1'last`	*15*
`tab2'range(2)`	*5..6*
`var2'length`	*10*

Les tableaux

◇ Exploitation d'un objet tableau

Si `o` est un tableau à n dimensions $d_1, \ldots, d_n$, alors :

- `o(`*e 1*`, ..., `*e n*`)` désigne la composante dont les coordonnées sont *e 1*, ..., *e n* (les *e k* sont des expressions dont les valeurs sont compatibles avec les bornes des dimensions).
- `o(`*e 1*`..`*e n*`)` désigne une **tranche** du tableau `o` à une dimension (toutes les composantes de *e 1* à *e n*)

  L'affectation entre tranches de tableaux de dimensions compatibles est possible.

Exemple :

```
type tab is array (1..5) of integer;
m : tab;

-- modification de la composante integer en (4)
m(4) := 3+m(3);

-- tranche de tableau
m(1..2) := m(3..4);

-- le compilateur gere correctement les cas problematiques
m(1..4) := m(2..5);
m(2..5) := m(1..4);
```

Les tableaux

◇ Tableau contraint / non contraint

– type tableau **contraint** : tous les objets de ce type ont les mêmes bornes pour chaque dimension.

– type tableau **non contraint** : chaque objet de ce type peut avoir ses propres bornes pour chaque dimension.

Tout objet tableau a nécessairement pour type un tableau contraint, sauf s'il est un paramètre d'unité (→ auquel cas le tableau peut être ou pas contraint).

Exemple :

```
s : string(1..20); -- préciser impérativement le domaine

t : constant string := "texte"; -- inutile ici : le compilateur le déduit

u : string := "texte"; -- Faux : pas de déduction pour les constantes

procedure test(x : string) is -- domaine adapté à chaque appel
    ...
end test;

test(s); -- ici le domaine est 1..20 pour le parametre x

test(t); -- là c'est 1..5
```


Les tableaux

◇ Initialisation/affectation d'un objet tableau

```
objet tableau := agrégat de tableau;
```

Un **agrégat de tableau** est un littéral de type tableau.
Chaque composante est valuée en la désignant :

– soit par sa position implicite dans l'agrégat (composante **positionnée**).

– soit en désignant explicitement sa position (composante **nommée**).

Exemple :

```
type tab is array (1..9) of natural;

-- initialisation avec agrégat positionné :
t1 : constant tab := (3, 5, 3, 3, 4, 3, 4, 3, 3);
-- même initialisation avec agrégat nommé :
t2 : tab := tab'(2 => 5, 5|7 => 4, others => 3);
-- initialisation avec agrégat mixte :
t3 : tab := (3, 5, 3, 5|7 => 4, others => 3);
```

▷ cas particulier des chaînes de caractères

Une chaîne de caractères est une forme simplifiée d'agrégat de tableau de caractères à une dimension :

Exemple :

Deux initialisations identiques :

```
txt1 : constant string := "chaine";
txt2 : constant string := ('c', 'h', 'a', 'i', 'n', 'e');
```


Les tableaux

◇ Sous-type d'un type tableau

Définir un sous-type de tableau, c'est restreindre les bornes des dimensions.

Exemple :

```
type tab is array (natural range <>) of integer;

subtype tab1 is tab(10..100);

subtype tab2 is tab1(tab1'first..20);
```

◇ Opérateur prédéfini (pour 1 dimension seulement)

Soient `a` et `b` deux valeurs de type tableau à une dimension dont les composantes sont de type T, et `x` une valeur de type T.

`a & b` concaténation des 2 tableaux
`a & x` ajout d'une nouvelle composante à droite
`x & a` ajout d'une nouvelle composante à gauche

Chacun des résultats est un tableau de dimension 1 dont les composantes sont de type T.

Exemple :

```
type tab is array (1..9) of integer;
t : tab := (2, 3, 4, 5, 6, 7, 8, 9, 1);

-- rotation 1 cran à gauche de toutes les composantes
t := t(2..9) & t(1);

-- idem, mais à droite
t := t(9) & t(1..8);

-- échange des composantes 1..4 avec 6..9
t := t(6..9) & t(5) & t(1..4);
```


Les tableaux

◇ Opérateurs prédéfinis sur tableaux booléens

Soient `a` et `b` deux valeurs d'un même type tableau dont les composantes sont de type `boolean`.

`not a`	**not** composante par composante
`a and b`	composante **and** composante
`a and then b`	idem avec **and then**
`a or b`	idem avec **or**
`a or else b`	idem avec **or else**
`a xor b`	idem avec **xor**

Les opérateurs retournent un tableau de même type, dont les composantes résultent de l'application du même opérateur mais sur les composantes elles-mêmes.

◇ Opérateurs prédéfinis sur tableaux discrets

Soient `a` et `b` deux valeurs d'un même type tableau dont les composantes sont de type discret.

`a < b`
`a <= b`
`a > b`
`a >= b`

Les opérateurs retournent un booléen ; la comparaison s'effectue composante par composante, dans l'ordre de définition des dimensions, et dans l'ordre des index pour chaque dimension.

Les types enregistrements

type Enregistrement = produit de types quelconques.

valeur Enregistrement = valeur décomposable en valeurs (**composantes**) de types quelconques.

Accès à une composante via un système de noms.

◇ Forme générale d'une définition

```
type nom is record
        liste de composantes
end record;
```

Une **composante** peut prendre l'une des formes suivantes :

```
liste de noms : type;
liste de noms : type := expression;
```

La deuxième forme permet d'initialiser des composantes → notion de **valeur par défaut**.

Exemple :

```
type complexe is record
       re, im : float := 0.0;
end record;

type date is record
       jour : natural range 1..31;
       mois : natural range 1..12;
      annee : natural range 1900..2100;
end record
```


Les enregistrements

◇ Attributs des types enregistrements

Si c est un nom de composante d'un type enregistrement, alors :

`c'position` = positive relative de c en unité mémoire

◇ Exploitation d'un objet enregistrement

Si o est un objet enregistrement dont une composante à pour nom m, alors $o.m$ désigne cette composante.

◇ Initialisation/affectation des objets enregistrements

```
objet enregistrement := agrégat d'enregistrement;
```

Un **agrégat d'enregistrement** est un littéral de type enregistrement. Chaque composante est valuée en la désignant :

– soit par sa position implicite dans l'agrégat (composante **positionnée**).

– soit en la nommant explicitement (composante **nommée**).

Les enregistrements

Exemple :

```
type un_mois is
     (Jan, Fev, Mar, Avr, Mai, Jun, Jul, Aou, Sep, Oct, Nov, Dec);

subtype une_annee is natural range 1900..2100;

subtype un_jour is natural range 1..31;

type date is record
      jour : un_jour;
      mois : un_mois;
     annee : une_annee;
end record;

-- quelques définitions d'objets

    -- initialisation par agrégat positionné
premier_Mai_93 : constant date := (1, Mai, 1993);

    -- initialisation par agrégat nommé
aujourdhui : date := (mois => Sep,
                      jour => 29,
                      annee => 1995);

-- quelques affectations

aujourdhui := premier_Mai_93;
aujourdhui.annee := aujourdhui.annee + 1;
```


Les types accès

Un type **accès** n'est autre qu'un type **pointeur**.

◇ Motivations

- représentation de **types récursifs** ; une valeur d'un tel type :
  - à une taille pouvant varier en cours d'exécution
  - cette taille peut, potentiellement, être infinie
- économie de place mémoire ; si n objets ont tous une même valeur v d'un type T, mieux vaut les coder comme n accès sur v.

◇ Forme générale d'une définition

```
type nom is access type;
```

Exemple :

Si ce type récursif était définissable, chacune de ses valeurs serait de taille infinie :

```
type liste is record
     element : integer;
     suivant : liste;
end record;
```

La notion d'accès permet de définir un tel type récursif :

```
type chainon;   -- déclaration d'un nom de type
type liste is access chainon;
type chainon is record
     element : integer;
     suivant : liste;
end record;
```

Les types accès

◇ Opérateur prédéfini

new *type* retourne un accès sur un nouvel objet de type *type* nécessairement contraint.

null est une constante définie par le compilateur pour chaque type accès défini.

◇ Exploitation d'un objet accès

Si p est un accès sur un objet v, alors p**.all** désigne v.

Exemple :

```
declare
     p : access integer := null;
     i : integer := 0;
begin
     p := new integer;
     p.all := i+1;
     i := p.all;
end;
```

Exemple :

Création d'une liste composée des trois éléments [1,2,3]:

```
declare
     l : liste := null;
begin
     l := new chainon'(element => 3, suivant => l);
     l := new chainon'(element => 2, suivant => l);
     l := new chainon'(element => 1, suivant => l);
end;
```


Les types paramétrés

◇ Forme générale d'une définition

```
type nom (paramètres) is record
        composantes
end record;
```

Un **paramètre** est de la forme :

```
liste de noms : type discret
liste de noms : type discret := expression
```

La deuxième forme permet de spécifier une valeur par défaut.
Un paramètre est considéré comme une composante.

Exemple :

```
type tableau is array (positive range <>, positive range <>)
     of float;

type matrice(l,c : positive := 10) is record
    elts : tableau(1..l, 1..c);
end record;
```


Les types paramétrés

◇ Type contraint / non contraint

un type enregistrement paramétré contraint vérifie au choix :

– aucun paramètre ne possède de valeur par défaut

– tout paramètre possédant une valeur par défaut est explicitement valué. Dans ce cas il n'est plus modifiable par affectation.

Un type non contraint est donc tel qu'au moins un paramètre n'est pas valué explicitement (usage de la valeur par défaut).

Exemple :

Un enregistrement paramétré contraint :

```
type vecteur (taille : integer) is record
     composantes : array (1..taille) of float;
end record;

-- toute définition d'objet doit valuer le paramètre
v1 : vecteur(3);  -- v1.taille est constant et vaut 3
v2 : vecteur(10);
```

Un enregistrement paramétré non contraint :

```
type vecteur (taille : integer := 10) is record
      composantes : array (1..taille) of float;
end record;

-- toute définition d'objet peut valuer le paramètre
v1 : vecteur(3); -- le type de l'objet est un type contraint
v2 : vecteur;    -- cette fois, il est non contraint
```

L'affectation d'un objet non contraint doit être effectuée **globalement** (i.e. pas composante par composante).

Les types paramétrés

◇ Attributs des types paramétrés

Si v est un objet d'un type paramétré, alors :

v**'constrained** = vrai si le type de v est contraint.

Exemple :

```
m1 : matrice;       -- m1'constrained = false
m2 : matrice(3, 4); -- m2'constrained = true
m3 : matrice(3);    -- m3'constrained = false
```

◇ Sous-type d'un type paramétré

Construire un sous-type d'un type paramétré consiste à valuer tous les paramètres non contraints.

Exemple :

```
type tableau is array (positive range <>, positive range <>)
     of float;

type matrice(l,c : positive := 10) is record
    elts : tableau(1..l, 1..c);
end record;

subtype tenseur is matrice(4, 4);
```

Les types enregistrements variables

Un type **enregistrement variable** est une forme particulière de type paramétré: au moins une composante est une somme de composantes.

◇ Forme générale d'une définition

```
type nom (paramètres) is record
        composantes
end record;
```

Un **paramètre** est de la même forme que celle définie pour les types paramétrés.

Une composante est soit :

– celle d'un type enregistrement banal

– variable :

```
case nom is
      when expr => composante
      ...
      when expr => composante
      when others => composante
end case;
```

où *nom* est le nom d'un des paramètres du type.

Toutes les composantes doivent avoir un nom différent.

Les types enregistrements variables

Exemple :

```
type genre_figure is (point, cercle, carre, triangle);
type longueur is new float range 0.0 .. float'last;

type figure(son_genre : genre_figure := point) is record
      sa_couleur : couleur;
      case son_genre is
         when    point => null;  -- aucune composante
         when   cercle => son_rayon : longueur;
         when    carre => son_cote  : longueur;
         when triangle => sa_hauteur, sa_base : longueur;
      end case;
end record;
```

Supposons définis les objets suivants :

```
o1 : figure;           -- objet non contraint
o2 : figure(carre);    -- objet contraint
o3 : figure(triangle); -- objet contraint
```

Quelques instructions :

```
o1.son_genre := cercle;  -- interdit : affectation globale seulement
o1 := (sa_couleur => rouge, son_genre => cercle, son_rayon => 1.0);
                                                          -- OK
o1 := (point, o1.sa_couleur);  -- OK

o2.son_genre := point;   -- interdit, objet contraint
o2 := (carre, vert, 1.0);-- OK
o2 := (point, rouge);    -- interdit, objet contraint

o3 := (son_genre  => triangle,
       sa_couleur => bleu,
       sa_base    => 2.0,
       sa_hauteur => 1.0);  -- OK
```


Les exceptions

◇ Forme générale de la déclaration

Dans une partie déclaration d'entités d'un bloc ou d'une unité

```
liste de noms d'exceptions : exception;
```

◇ Forme générale du traitement

Dans le corps d'un bloc ou d'une unité:

```
begin
      instructions du corps
exception
      when liste de noms => séquence d'instructions
      ...
      when liste de noms => séquence d'instructions
      when others => séquence d'instructions
end;
```

Une liste de noms est composée d'un ou plusieurs (dans ce cas séparés par le caractère |) noms d'exception.

◇ Déclenchement d'une exception

Instruction **raise** :

```
raise nom d'exception;
raise;
```

Deuxième forme autorisée seulement dans un traitement d'exception.

Sémantique des exceptions

◇ Principe général sans les tâches

Lorsqu'une exception est levée (instruction **raise**) dans un bloc (ou corps d'unité) :

→ recherche au niveau du bloc du cas d'exception traitant l'exception levée.

Si cette recherche échoue, l'exécution du bloc courant est avortée, et la même exception est levée dans le bloc appelant (englobant).

→ sinon, exécution de la séquence d'instructions correspondante ; deux cas se présentent :

⇒ aucune instruction **raise** n'a été exécutée durant le traitement : l'exécution du bloc courant se termine normalement → le bloc appelant continue normalement.

⇒ une exception a été levée : l'exécution du bloc courant avorte → l'exception est levée dans le bloc appelant.

◇ Principe général avec les tâches

Un programme sans tâche explicitement déclarée n'est composé que d'une seule tâche.

Si plusieurs tâches, le principe reste correct, sauf que la remontée au bloc appelant s'arrête à la tâche → une tâche ne peut pas lever directement une exception dans une autre tâche, même celle qui lui a donné naissance.

Sémantique des exceptions

Exemple :

```
declare
    a, b : exception;

    procedure P(i : integer) is
    begin
        if i=0
            then raise a;
            else P(i-1);
        end if;
    exception
        when a => put(i);
                  raise;  -- ©
    end P;

begin
    P(5);
exception
    when a => put_line("C'est fini");
    when b => put_line("OK");
end;
```

À l'exécution, il est affiché :

```
0 1 2 3 4 5 C'est fini
```

Si l'on remplace © *par* **raise** b, *alors s'affiche :*

```
0 OK
```


Les paquetages

◇ Motivations

- regrouper des entités ADA (objets, types, unités, exceptions, tâches) en vue d'une ré-utilisation.
- offrir des fontionnalités (ou services) dont les détails d'implantation en ADA sont cachés (→ types abstraits).

  Assurance que l'usager sera dans l'incapacité de réaliser des opérations remettant en cause le bon fonctionnement de ces services.

◇ Généralités

- **paquetage** est la terminologie ADA pour désigner un **module** dans les autres langages de programmation.
- un paquetage est défini par :
  - une partie **spécification** → notion d'**interface** de module.
  - éventuellement une partie **corps** (*body*) → notion d'**implantation** de module.

Spécification des paquetages ▷

◇ Description

Une **spécification** de paquetage décrit les entités ADA qu'il offre ; les choix d'implantation peuvent être cachés.

◇ Forme générale

package *nom* **is**
    déclarations d'entités publiques
private
    déclarations d'entités privées
end *nom*;

La partie **private** *déclarations d'entités privées* est optionnelle.

Exemple :

```
package noms_du_calendrier is
    type jour is (lun, mar, mer, jeu, ven, sam, dim);
    type mois is (jan, fev, mar, avr, mai, jun, jul, aou, sep,
                  oct, nov, dec);
    subtype annee is natural range 1990 .. 2100;

    mois_Noel : constant mois := dec;
end noms_du_calendrier;
```


◁ Spécification des paquetages ▷

◇ Entités à représentation cachée

La partie publique d'un paquetage peut définir des entités dont l'implantation est cachée → la partie **private** contient alors certaines implantations.
Peuvent être cachées les définitions :

– de sous-unités → simples déclarations

  Les définitions sont alors décrites dans le corps du paquetage.

– de constantes → notion de constante **différée**

  Les définitions sont alors décrites dans la partie **private** de la spécification du paquetage.

– de types → notion de type **privé**

  Les définitions sont alors décrites dans la partie **private** de la spécification du paquetage.

▷ Constante différée

Exemple :

```
package exemple is
    un, one, ein : constant integer;
private
    un, one, ein : constant integer := 1;
end exemple;
```

Spécification des paquetages

▷ Types privés

Un type privé est tel que toutes ses valeurs doivent être considérées par l'usager comme un tout indécomposable.

type *type* **is private;**

Exemple :

Une version sans cacher les définitions :

```
package les_complexes_1 is
   type complexe is record
       re, im : float := 0.0;
   end record;
   i : constant complexe := (0.0, 1.0);
end les_complexes_1;
```

Une autre version cachant les définitions :

```
package les_complexes_2 is
   type complexe is private;
   i : constant complexe;
private
   type complexe is record
       re, im : float := 0.0;
   end record;
   i : constant complexe := (0.0, 1.0);
end les_complexes_2;
```

Seules opérations autorisées à l'usager sur un type privé :

– le passage de paramètres pour des unités qu'il définit

– l'affectation lorsqu'il s'agit d'une variable

– les comparateurs **=** et **/=**

– les opérations offertes par le paquetage

Spécification des paquetages

▷ Types privés limités

Un type privé limité est un type privé dont les seules opérations autorisées à l'usager sont :

– le passage de paramètres sans valeur par défaut pour des unités qu'il définit

– les opérations offertes par le paquetage

type *nom* **is limited private;**

Exemple :

```
package les_fichiers is
    type fichier is limited private;
    procedure ouvrir(f:in out fichier;n:string);
    function ouvert(f:fichier) return boolean;
    procedure fermer(f:in out fichier);
    ...
private
    type fichier is ...
    ...
end les_fichiers;
```

▷ Remarque sur les types privés

Si un type privé est paramétré, son paramétrage doit figurer dans la partie publique du paquetage.

Spécification des paquetages

◇ Élaboration d'une spécification

Élaborer une spécification de paquetage, c'est :

– élaborer toutes les entités de sa partie publique, en exploitant si nécessaire les informations de la partie **private**

Après élaboration de la spécification d'un paquetage p, toute entité e de sa partie publique est exploitable sous la forme $p.e$

Exemple :

```
declare
    type t is (un, deux);

    a, b : t := un;

    package exemple is
       subtype t is boolean;
       a, e : t := true;
    end exemple;

    c : t         := a;
    d : exemple.t := exemple.e;
begin
    ...
end;
```


Spécification des paquetages

◇ La clause d'utilisation use

La notation pointée permettant l'accès aux entités publiques d'un paquetage peut être évitée en faisant usage d'une **clause d'utilisation** du paquetage :

use *nom*;

Toute entité publique du paquetage peut alors être nommée directement → surcharge possible de noms existants → problèmes d'ambiguïté possibles.

Exemple :

```
bloc : declare
    type t is (un, deux);

    a, b : t := un;

    package exemple is
       subtype t is boolean;
       a, e : t := true;
    end exemple;

    use exemple;

-- ambiguïté sur c : t         := a;
    c : bloc.t    := bloc.a;
    d : exemple.t := e;
begin
    ...
end;
```

Une clause **use** ne s'applique qu'à un paquetage dont la spécification a déjà été élaborée.

Corps des paquetages

◇ Description

Un **corps** de paquetage définit l'implantation des sous-unités décrites dans sa spécification.

La portée des entités déclarées / définies dans la spécification d'un paquetage s'étend au corps de ce paquetage.

Pour ses besoins propres, le corps peut définir des entités qui ne seront exploitables que par lui seul.

◇ Forme générale

package body *nom* **is**
    déclarations d'entités
begin
    instructions
exception
    traitement d'exception
end *nom*;

Sont optionnelles les parties :

– *déclarations d'entités*

– **exception** *traitement d'exception*

Corps de paquetages

◇ Élaboration d'un corps de paquetage

Élaborer un corps de paquetage, c'est :

– élaborer toutes les entités de sa partie déclarative

– exécuter les instructions de son corps principal, avec traitement éventuel des exceptions levées durant cette exécution.

Clause de dépendance

Toute définition d'unité devant exploiter un paquetage non encore élaboré doit être précédée d'une **clause de dépendance** :

with *liste de noms de paquetages*;
définition de l'unité

Si les entités du paquetage doivent être exploitées directement par l'unité, faire suivre cette clause d'une clause d'utilisation;

with *liste de noms de paquetages*; **use** *idem*;
définition de l'unité

Exemple :

```
package exemple is
   type t is (un, deux);
   a, b : t := un;
end exemple;

with exemple; use exemple;
procedure test is
   c : t;
begin
   ...
end test;
```

C'est un cas particulier d'exploitation de cette clause (voir la section sur la compilation séparée, page 137).

Un exemple de paquetage

◇ Nombres complexes : rien n'est caché

Exemple :

La spécification du paquetage :

```
package les_complexes is
     type complexe is record
          re, im : float := 0.0;
     end record;

     i : constant complexe := (0.0, 1.0);

     function "+"(x, y : complexe) return complexe;
     function "*"(x, y : complexe) return complexe;
end les_complexes;
```

Son corps :

```
package body les_complexes is

  function "+"(x, y : complexe) return complexe is
  begin
       return (x.re+y.re, x.im+y.im);
  end "+";

  function "*"(x, y : complexe) return complexe is
  begin
       return (x.re*y.re-x.im*y.im, x.re*y.im+x.im*y.re);
  end "*";

begin
  null;
end les_complexes;
```

Un exemple de paquetage

Exemple :

Une procédure exploitant ce paquetage :

```
with les_complexes; use les_complexes;
procedure exemple is

   a, b : complexe := (0.0, 0.0);

begin
   a.re := 1.0;  -- possible : on sait ce qu'est complexe
   if b /= i then
      b := b+a*i;
   end if;
end exemple;
```


Un exemple de paquetage

◇ Nombres complexes : tout est caché

Exemple :

La spécification du paquetage :

```
package les_complexes is
     type complexe is private;
     i : constant complexe;

     function reel(x:complexe) return float;
     function imag(x:complexe) return float;
     function cplx(r,i:float) return complexe;

     function "+"(x, y : complexe) return complexe;
     function "*"(x, y : complexe) return complexe;
private
     type complexe is record
         re, im : float := 0.0;
     end record;
     i : constant complexe := (0.0, 1.0);
end les_complexes;
```


Un exemple de paquetage

Exemple :

Son corps :

```
package body les_complexes is

  function reel(x:complexe) return float is
  begin
      return x.re;
  end reel;

  function imag(x:complexe) return float is
  begin
      return x.im;
  end imag;

  function cplx(r, i:float) return complexe is
  begin
      return (r, i);
  end cplx;

  function "+"(x, y : complexe) return complexe is
  begin
      return (x.re+y.re, x.im+y.im);
  end "+";

  function "*"(x, y : complexe) return complexe is
  begin
      return (x.re*y.re-x.im*y.im, x.re*y.im+x.im*y.re);
  end "*";

begin
  null;
end les_complexes;
```


Un exemple de paquetage

Exemple :

Une procédure exploitant ce paquetage :

```
with les_complexes; use les_complexes;
procedure exemple is

-- impossible d'écrire  a, b : complexe := (0.0, 0.0);
-- en effet, on ne sait pas ce qu'est complexe
        a, b : complexe; -- OK
begin
-- impossible d'écrire  a.re := 1.0;
   a := cplx(1.0, imag(a)); -- une solution possible
   if b /= i then
      b := b+a*i;
   end if;
end exemple;
```

Les paquetages prédéfinis

◇ Paquetage implicitement exploité

Le paquetage **standard** est implicitement exploité dès qu'une unité est décrite.

◇ Les autres paquetages

`system` : ce qui est spécifique à la machine hôte

`text_io` : tout sur les entrées sorties

`calendar` : tout le nécessaire pour manipuler les dates

etc

Les unités génériques

◇ Intérêt de la généricité en Ada

– passage de sous-programmes (fonction ou procédure) en paramètre

– variabiliser les types des paramètres (→ polymorphisme de type)

Exemple :

Pour illustrer le polymorphisme de type (i.e. les types peuvent être variabilisés), considérons la fonction calculant le `min` *de deux valeurs :*

```
function min(x, y : integer) return integer is
begin
    if x < y then return x; else return y; end if;
end min;

function min(x, y : float) return float is
begin
    if x < y then return x; else return y; end if;
end min;
```

Ici `min` *est seulement définie pour les types* `integer` *et* `float`*. Nous savons que, quel que soit le type* ***T****, cette fonction aura toujours la forme suivante :*

```
function min(x, y : T) return T is
begin
    if x < y then return x; else return y; end if;
end min;
```

La définition suppose que l'opérateur `<` *est défini sur* ***T****.*

Les unités génériques

◇ Forme générale

▷ Déclaration d'une unité générique

```
generic
    paramètres de généricité
déclaration de l'unité;
```

▷ Définition d'une unité générique

– toute définition d'unité générique impose que l'unité ait été **obligatoirement** déclarée au préalable.

– les paramètres **ne doivent pas** être redéfinis lors de la définition de l'unité.

Exemple :

```
generic
    type T is private;
procedure echange(x, y : in out T);

procedure echange(x, y : in out T) is
    z : T := x;
begin
    x := y;
    y := z;
end echange;
```

Le paramètre de généricité `T` *dénote ici un type.*

Les paramètres de généricité

◇ Paramètres d'objet classiques

Du genre de ceux des procédures :

```
nom : mode type valeur par défaut;
```

où *mode* est optionnel, de la forme **in**, **in out** ou **out**

ainsi que *valeur par défaut*, de la forme `:=` *expression*.

◇ Paramètres de sous-programme

```
with function nom entete return type;
with procedure nom entete;
```

Le paramètre *nom* désigne dans l'unité le nom d'un sous-programme. Doit **impérativement** être instancié par un nom de sous-programme satisfaisant les contraintes indiquées.

```
with function nom entete return type is nom';
with procedure nom entete is nom';
```

Idem, mais *nom* peut ne pas être instancié explicitement, auquel cas il sera instancié par *nom'* (→ similaire à une valeur par défaut).

```
with function nom entete return type is <>;
with procedure nom entete is <>;
```

Idem, mais le nom par défaut pour l'instanciation sera *nom*.

Les paramètres de généricité

◇ Paramètres de type

type *T* is private;

T instanciable par tout type ; seules opérations réalisables sur les objets de l'unité ayant ce type : affectation, tests d'égalité (= et /=), définition / passage de paramètres.

type *T* is limited private;

T instanciable par tout type ; seule opération réalisable sur les objets de l'unité ayant ce type : définition / passage de paramètres.

type *T* is (<>);

T instanciable par tout type discret.

type *T* is range <>;

T instanciable par tout type entier.

type *T* is digits <>;

T instanciable par tout type réel flottant.

type *T* is delta <>;

T instanciable par tout type réel fixe.

Les paramètres de généricité

type *T* is array (*dimensions*) of *T'*;

T instanciable par tout type tableau ayant les contraintes indiquées (dimensions, composantes de type *T'*).

type *T* is access *T'*;

T instanciable par tout type accès sur *T'*.

Exemples d'unités génériques

Exemple :

Une procédure générique capable de trier (selon l'algorithme du tri à bulles) tout tableau de dimension 1 :

```
generic
    type E is private;
    type I is (<>);
    type A is array(I) of E;
    with function ">"(x, y:E) return boolean is <>;
procedure tri_bulle(t : in out A);

procedure tri_bulle(t : in out A) is

    procedure echange(x, y:in out E) is
       z : E := x;
    begin
       x := y;
       y := z;
    end echange;

begin
    for m in reverse I'succ(t'first)..I'pred(t'last) loop
        for n in t'first..m loop
            if t(n) > t(I'succ(n)) then
               echange(t(n), t(I'succ(n)));
            end if;
        end loop;
    end loop;
end tri_bulle;
```


Exemples d'unités génériques

Exemple :

Un paquetage offrant des services de manipulation de piles de taille bornée.

```
generic
    taille_max : natural;
    type E is private;
package piles_bornees is
    type pile_bornee is private;

    pile_vide, pile_pleine : exception;

    procedure empiler(p : in out pile_bornee; x : E);
    procedure depiler(p : in out pile_bornee; x : out E);

private
    type table is array (positive range <>) of E;
    type pile_bornee is record
        sommet   : natural range 0..taille_max := 0;
        elements : table(1..taille_max);
    end record;
end piles_bornees;

package body piles_bornees is
    procedure empiler(p : in out pile_bornee; x : E) is
    begin
        if p.sommet = taille_max then raise pile_pleine; end if;
        p.sommet             := p.sommet + 1;
        p.elements(p.sommet) := x;
    end empiler;

    procedure depiler(p : in out pile_bornee; x : out E) is
    begin
        if p.sommet = 0 then raise pile_vide; end if;
        x        := p.elements(p.sommet);
        p.sommet := p.sommet - 1;
    end depiler;
end piles_bornees;
```

Instanciation

◇ Instanciation d'une unité générique

Une unité générique ne peut être exploitée telle quelle : tous ses paramètres de généricité doivent avoir été **instanciés** :

- tout paramètre d'objet doit être lié à une valeur (→ mode **in**) ou à un nom de variable (→ modes **out** et **in out**).
- tout paramètre de type doit être lié à un type satisfaisant les contraintes du paramètre.
- tout paramètre de sous-programme doit être lié à un nom de sous-programme satisfaisant les contraintes du paramètre.

package *nom* **is new** *nom pag. gén.* *instanciations*;
function *nom* **is new** *nom fonc. gén.* *instanciations*;
procedure *nom* **is new** *nom proc. gén.* *instanciations*;

Exemple :

```
generic
    type T is private;
procedure echange(x, y : in out T);
procedure echange(x, y : in out T) is
    z : T := x;
begin
    x := y; y := z;
end echange;

procedure echange_integer is new echange(integer);
procedure echange_float is new echange(float);
```

Créer 2 spécialisations de la procédure générique `echange` *où* `T` *a été remplacé par* `integer` *puis par* `float`.

Instanciation

Exemple :

La procédure générique de tri d'un tableau était déclarée ainsi :

```
generic
    type E is private;
    type I is range <>;
    type T is array(I) of E;
    with function ">"(x, y:E) return boolean is <>;
procedure tri_bulle(t : in out T);
```

Un exemple d'exploitation de cette procédure :

```
procedure exemple is

    type un_element is (un, deux, trois, quatre, cinq);

    subtype un_indice is natural range 1 .. 10;

    type un_tab is array (un_indice) of un_element;
    tab : un_tab := (un, trois, deux, cinq, others => quatre);

    procedure mon_tri is
              new tri_bulle(un_element, un_indice, un_tab);

begin
    mon_tri(tab);
end exemple;
```


Instanciation

Exemple :

Exploitation du paquetage des piles bornées :

```
with piles_bornees;
procedure exemple is

    package pb_int_100 is new piles_bornees(100, integer);
    use pb_int_100;

    pb1 : pile_bornee;

    package pb_bool_10 is new piles_bornees(10, boolean);
    use pb_bool_10;

    -- à partir de maintenant, 2 types pile_bornee
    pb2 : pb_int_100.pile_bornee;
    pb3 : pb_bool_10.pile_bornee;

begin
    empiler(pb1, 10);
    empiler(pb1, 20);

    empiler(pb3, true);
    empiler(pb3, false);
end exemple;
```


Les entrées/sorties

◇ Deux formes d'entrées-sorties

→ sous forme binaire

- les fichiers à accès séquentiel
- les fichiers à accès direct

→ sous forme textuelle

- les fichiers textes

◇ Les exceptions d'entrées-sorties

```
package io_exception is
    status_error,
      -- action sur fichier non ouvert, ou inapplicable à un fichier ouvert
    mode_error,   -- action incompatible avec le mode d'usage du fichier
    name_error,   -- mauvais nom de fichier
    use_error,    -- erreur liée au système d'exploitation
    device_error, -- erreur liée au périphérique (disque plein, ...)
    end_error,    -- fin de fichier dépassée
    data_error,   -- donnée lue du mauvais type
    layout_error  -- erreur de manipulation de page, ligne ou colonne
                  : exception;
end io_exception;
```

Les fichiers séquentiels

◇ Le paquetage concerné

Le paquetage générique `SEQUENTIAL_IO`.

◇ En résumé

Le fichier est composée de **fiches** (ou **articles**, **enregistrements**) ayant toutes un même type pouvant être choisi à volonté (paramètre de généricité → variable de type).

L'accès aux fiches est séquentiel : un **curseur** indique la fiche courante.

A l'ouverture d'un fichier séquentiel, le curseur est positionnée sur la première fiche ; toute opération d'accès (lecture ou écriture) incrémente automatiquement le curseur d'une unité.

Le paquetage SEQUENTIAL_IO

```
with io_exceptions;
generic
    type element_type is private;
```

Toutes les fiches ont un même type générique `element_type`.

```
package sequential_io is

    type file_type is limited private;
    type file_mode is (in_file, out_file);
```

Un fichier séquentiel est un objet du type `file_type`. *Il ne peut être ouvert qu'en lecture seule (*`in_file`*) ou écriture seule (*`out_file`*).*

```
    procedure create(file : in out file_type;
                     mode : file_mode := out_file;
                     name : string := "";
                     form : string := "");
```

Création d'un nouveau fichier puis l'ouvre; `form` *indique les propriétés système du fichier (attributs de protection, etc) ; un* `name` *vide indique un fichier temporaire.*

```
    procedure open(file : in out file_type;
                   mode : file_mode;
                   name : string;
                   form : string := "");
```

Ouvre un fichier déjà créé et non encore ouvert. Le curseur des fiches vaut alors 0.

```
    procedure close(file : in out file_type);
```

Fermeture d'un fichier ouvert.


```
    procedure delete(file : in out file_type);
```

Fermeture puis effacement d'un fichier ouvert.

```
    procedure reset(file : in out file_type);
    procedure reset(file : in out file_type ;
                    mode : file_mode);
```

Ré-ouvre un fichier ouvert (en changeant de mode éventuellement). Le curseur est ramené à 0.

```
    function mode(file : file_type) return file_mode;
    function name(file : file_type) return string;
    function form(file : file_type) return string;
```

Fonctions permettant de retrouver les paramètres donnés à l'ouverture d'un fichier.

```
    function is_open(file : file_type) return boolean;
```

Teste si le fichier est déjà ouvert.

```
    function end_of_file(file : file_type) return boolean;
```

Teste si le curseur vient de passer la dernière fiche ; accessible uniquement en mode `in_file`.

```
    procedure read(file : in out file_type ;
                   item : out element_type);
```

Lit la fiche courante (désignée par le curseur) et l'affecte à `item` *; le curseur passe automatiquement à la fiche suivante.*

```
    procedure write(file : in out file_type ;
                    item : element_type);
```

Écrit la valeur de `item` *sur la fiche courante; le curseur passe automatiquement à la fiche suivante.*

```
private
        -- dépendant du système
end sequential_io;
```


Un exemple d'exploitation

Exemple :

```
with sequential_io;
procedure exemple is
    type complexe is record
        re, im : float;
    end record;

    package fs_complexe is new sequential_io(complexe);
        use fs_complexe;

    fic  : file_type;
    nom  : constant string := "test.seq";

    nmax : constant := 10;
    tab  : array (1..nmax) of complexe;

begin
    create(fic, out_file, nom);
    for i in tab'range loop
        tab(i) := (re => float(i), im => float(-i));
        write(fic, tab(i));
    end loop;
    close(fic);

    tab := (others => (0.0, 0.0));

    open(fic, in_file, nom);
    declare
        i : integer := tab'first;
    begin
        while not end_of_file(fic) loop
            read(fic, tab(i));
            i := i+1;
        end loop;
    end;
    close(fic);
end exemple;
```

Un exemple d'exploitation

Exemple :

Si l'on souhaite créer le fichier seulement quand il n'existe pas, sinon ouvrir celui qui existe déjà:

```
...
begin
    begin
        open(fic, in_file, nom);
        put_line("Le fichier " & nom & " existe de'ja`.");
    exception
      when fs_complexe.name_error =>
        create(fic, out_file, nom);

        for i in tab'range loop
            tab(i) := (re => float(i), im => float(-i));
            write(fic, tab(i));
        end loop;

        close(fic);

        put_line("Le fichier " & nom & " vient d'e^tre cre'e'.");

        open(fic, in_file, nom);
    end;

    tab := (others => (0.0, 0.0));

    declare
        i : integer := tab'first;
    begin
        ...
```


Les fichiers directs

◇ Le paquetage concerné

Le paquetage générique `DIRECT_IO`.

◇ En résumé

Le fichier est composée de fiches ayant toutes un même type pouvant être choisi à volonté (paramètre de généricité → variable de type).

L'accès aux fiches est quelconque : toute fiche possède un index, et l'index est positionnable à volonté dans l'intervalle `1 .. `***`size`***, où ***`size`*** est le plus grand index manipulé jusqu'ici (procédure `set_index`).

À chaque index ne correspond pas forcément une fiche (l'espace mémoire est réservé, mais n'est pas utilisé).

À l'ouverture d'un fichier, l'index est positionnée sur la première fiche ; toute opération d'accès (lecture ou écriture) incrémente automatiquement l'index d'une unité.

Le paquetage DIRECT_IO

```
with io_exceptions;
generic
    type element_type is private;
```

Toutes les fiches ont un même type générique `element_type`.

```
package direct_io is
    type file_type is limited private;
    type file_mode is (in_file, out_file, inout_file);
    type count is range 0..dépendant du système;
    subtype positive_count is count range 1..count'last;
```

Un fichier direct est un objet du type `file_type`. *Il peut être ouvert en lecture seule (*`in_file`*), en écriture seule (*`out_file`*), ou les deux (*`inout_file`*). L'index courant est un entier naturel du type* `count` *; toute fiche a nécessairement un index positif de type* `positive_count`.

```
    procedure create(file : in out file_type;
                     mode : file_mode := inout_file;
                     name : string := "";
                     form : string := "");

    procedure open(file : in out file_type;
                   mode : file_mode;
                   name : string;
                   form : string := "");

    procedure close(file : in out file_type);

    procedure delete(file : in out file_type);
```

Voir les pages 113 et 114.


```
    procedure reset(file : in out file_type);
    procedure reset(file : in out file_type ;
                    mode : file_mode);

    function mode(file : file_type) return file_mode;
    function name(file : file_type) return string;
    function form(file : file_type) return string;

    function is_open(file : file_type) return boolean;
    function end_of_file(file : file_type) return boolean;
```

Voir la page 114.

```
    function index(file : file_type) return positive_count;
```

Retourne l'index de la fiche courante.

```
    function size(file : file_type) return count;
```

Retourne le plus grand index du fichier.

```
    procedure set_index(file : file_type ;
                        to : positive_count);
```

Positionne l'index à la valeur `to` *indiquée ; peut dépasser* `size(...)` → *deviendra son résultat si appel.*

```
    procedure read(file : in out file_type ;
                   item : out element_type);
    procedure read(file : in out file_type ;
                   item : out element_type ;
                   from : positive_count);
```

Lit la fiche courante ou d'index `from` *et l'affecte à* `item`. *Exception* `data_error` *si fiche jamais écrite.*

```
procedure write(file : in out file_type ;
                item : element_type);
procedure write(file : in out file_type ;
                item : element_type ;
                to   : positive_count);
```

Écrit la valeur de `item` *sur la fiche courante ou d'index* `to`.

```
private
    -- dépendant du système
end direct_io;
```


Un exemple d'exploitation

Exemple :

```
with direct_io;
procedure exemple is
    type complexe is record
        re, im : float;
    end record;

    package fd_complexe is new direct_io(complexe);
         use fd_complexe;

    fic  : file_type;
    nom  : constant string := "test.dir";

    nmax : constant := 10;
    subtype dom is positive_count range 1 .. nmax;
    tab  : array (dom) of complexe;
    idx  : constant array (dom) of dom
         := (3, 2, 5, 1, 9, 4, 8, 6, 10, 7);

begin
    create(fic, out_file, nom);
    for i in tab'range loop
        tab(idx(i)) := (re => float(i), im => float(-i));
        write(fic, tab(idx(i)), idx(i));
    end loop;
    close(fic);

    tab := (others => (0.0, 0.0));

    open(fic, in_file, nom);
    for i in reverse tab'range loop
        read(fic, tab(idx(i)), idx(i));
    end loop;
    close(fic);
end exemple;
```


Les entrées-sorties textuelles ▷

◇ PUT et GET

Le paquetage `TEXT_IO` offre des entrées-sorties pour :

– les types de `character` et `string`

– tout type entier

– tout type énuméré

– tout type réel (fixe ou flottant)

Texte d'une valeur = le littéral `ADA` correspondant.

▷ GET

Permet l'entrée d'information.

Source des caractères : un fichier texte, le canal d'entrée standard (fichier texte par défaut) ou un objet de type `string`.

Tout caractère séparateur (blanc, tabulation, fin de ligne) est ignoré.

Littéral entré syntaxiquement incorrect → exception `data_error` levée ; dans ce cas, le curseur de lecture **n'est pas** avancé (caractères non consommés), permettant leur traitement ultérieur.

▷ PUT

Permet la sortie d'information.

Cible des caractères : un fichier texte, le canal de sortie standard (fichier texte par défaut) ou un objet de type `string`.

◁ Les entrées-sorties textuelles

◇ Organisation d'un fichier texte

– fichier texte = séquence de **pages**.

Le nombre de pages peut être quelconque.

– page = séquence de **lignes**.

Chaque page est numérotée (à partir de 1), se termine par un caractère spécial (fin de page).

Le nombre maximal de lignes pour toutes les pages peut être borné ou non (procédure `set_page_length`).

– ligne = séquence de caractères disposés en **colonnes**.

Chaque ligne est numérotée (à partir de 1), se termine par un caractère spécial (fin de ligne).

Le nombre maximal de colonnes pour toutes les lignes peut être borné ou non (procédure `set_line_length`).

– chaque colonne est numérotée (à partir de 1)

Le paquetage TEXT_IO

```
with io_exceptions;
package TEXT_IO is
    type file_type is limited private;
    type file_mode is (in_file, out_file);

    type count is range 0..dépendant du système;
    subtype positive_count is count range 1..count'last;
    unbounded : constant count := 0;

    subtype field is integer range 0..dépendant du système;
    subtype number_base is integer range 2 .. 16;

    sous-prog. de manipulation des fichiers textes
    sous-prog. de manipulation des canaux standard d'E/S
    sous-prog. de manipulation pages, lignes, colonnes

    sous-prog. d'E/S sur caratères et chaînes
    paquet. gén. d'E/S sur les énumérations : ENUMERATION_IO
    paquet. gén. d'E/S sur les entiers : INTEGER_IO
    paquet. gén. d'E/S sur les flottants : FLOAT_IO
    paquet. gén. d'E/S sur les fixes : FIXED_IO
private
    -- dépendant du système
end TEXT_IO;
```


Manipulation des fichiers textes

```
procedure create(file : in out file_type;
                 mode : file_mode := out_file;
                 name : string := "";
                 form : string := "");

procedure open(file : in out file_type;
               mode : file_mode;
               name : string;
               form : string := "");

procedure close(file : in out file_type);

procedure delete(file : in out file_type);

procedure reset(file : in out file_type);
procedure reset(file : in out file_type ;
                mode : file_mode);

function mode(file : file_type) return file_mode;
function name(file : file_type) return string;
function form(file : file_type) return string;

function is_open(file : file_type) return boolean;

function end_of_file(file : file_type) return boolean;
```

Voir les pages 113 et 114

Canaux standard d'E/S

```
procedure set_input(file : file_type);
procedure set_output(file : file_type);
```

Pour positionner le canal d'entrée ou de sortie standard sur un fichier texte.

```
function standard_input return file_type;
function standard_output return file_type;
```

Fichier texte représentant les canaux standard d'entrée ou de sortie du système d'exploitation ; par exemple, le canal d'entrée par défaut sous Unix est soit le terminal (clavier), soit un pipe*, soit un fichier (redirections des entrées), etc.*

```
function current_input return file_type;
function current_output return file_type;
```

Pour récupérer le fichier texte exploité comme canal d'entrée ou de sortie standard.

```
function end_of_file return boolean;
```

Teste si le canal d'entrée est toujours producteur de caractères.

Pages, lignes, colonnes

◇ Les pages

```
procedure set_page_length(file : file_type ; to : count);
procedure set_page_length(to : count);
```

Spécification du nombre de lignes max par page du fichier texte spécifié ou du canal courant. Si 0 ou unbounded*, pas de limitation.*

```
function page_length(file : file_type) return count;
function page_length return count;
```

Permet de retrouver le nombre de lignes max.

```
procedure new_page(file : file_type);
procedure new_page;
```

Engendre une nouvelle page après la page courante du fichier texte.

```
procedure skip_page(file : file_type);
procedure skip_page;
```

Ignore tous les caractères de la page courante → curseur sur page suivante.

```
function end_of_page(file : file_type) return boolean;
function end_of_page return boolean;
```

Teste si le curseur est en fin de page.

```
function page(file : file_type) return positive_count;
function page return positive_count;
```

Retourne le numéro de la page courante.

Pages, lignes, colonnes

◇ Les lignes

```
procedure set_line_length(file : file_type ; to : count);
procedure set_line_length(to : count);
```

Spécification du nombre de colonnes max par ligne du fichier texte spécifié ou du canal courant. Si nul, pas de limitation.

```
function line_length(file : file_type) return count;
function line_length return count;
```

Permet de retrouver le nombre de colonnes max.

```
procedure new_line(file : file_type);
procedure new_line;
```

Engendre une nouvelle ligne dans la page courante.

```
procedure skip_line(file : file_type);
procedure skip_line;
```

Ignore les caractères de la ligne courante → se déplace à la ligne suivante.

```
function end_of_line(file : file_type) return boolean;
function end_of_line return boolean;
```

Teste si le curseur est en fin de ligne.

```
procedure set_line(file : file_type ; to : positive_count);
procedure set_line(to : positive_count);
```

La ligne courante devient celle spécifiée.

```
function line(file : file_type) return positive_count;
function line return positive_count;
```

Retourne le numéro de la ligne courante.

Pages, lignes, colonnes

◇ Les colonnes

```
procedure set_col(file : file_type ; to : positive_count);
procedure set_col(to : positive_count);
```

La colonne courante devient celle spécifiée.

```
function col(file : file_type) return positive_count;
function col return positive_count;
```

Retourne le numéro de la colonne courante.

E/S des caractères et chaînes

◇ Les caractères

```
procedure get(file : file_type ; item : out character);
procedure get(item : out character);
```

Affecte à `item` *le prochain caractère à lire. Les séparateurs sont ignorés (blanc, tabulation, fin de ligne, fin de page).*

```
procedure put(file : file_type ; item : character);
procedure put(item : character);
```

Sortie du caractère `item`*; si nécessaire, passe à la ligne, voire à la page, suivante.*

◇ Les chaînes

```
procedure get(file : file_type ; item : out string);
procedure get(item : out string);
```

Affecte à `item` *les premiers caractères à lire jusqu'au premier séparateur rencontré (blanc, tabulation, fin de ligne, fin de page, fin de fichier).*

```
procedure put(file : file_type ; item : string);
procedure put(item : string);
```

Sortie de tous les caractères de `item`*; si nécessaire, passe à la ligne, voire à la page, suivante.*


```
procedure get_line(file : file_type ;
          item : out string ;
          last : out natural);
procedure get_line(item : out string ;
          last : out natural);
```

Affecte à `item` *tous les caractères à lire jusqu'à la fin de la ligne.* `last` *contient l'index du dernier caractère stocké dans* `item` *(le premier est stocké en* `item(item'first)`*). Appel ensuite* `skip_line` *→ passe à la ligne suivante.*

```
procedure put_line(file : file_type ; item : string);
procedure put_line(item : string);
```

équivalent à `put(...)` *suivi de* `new_line`*.*

E/S des types énumérés

```
generic
    type enum is (<>);
```

Le paquetage est valable pour tout type énuméré `enum`

```
package enumeration_io is
    procedure get(file : file_type ;
                  item : out enum);
    procedure get(item : out enum);
    procedure get(from : string ;
                  item : out enum ;
                  last : out natural);
```

Cherche à interpréter les prochains caractères à lire comme l'un des littéraux du type. La source des caractères peut être un fichier, le canal d'entrée courant ou une chaîne de caractères ; dans ce dernier cas, `last` *contient la position du dernier caractère constituant le littéral.*

```
    procedure put(file  : file_type ;
                  item  : enum ;
                  width : field := 0 ;
                  set   : type_set := upper_case);
    procedure put(item  : enum ;
                  width : field := 0 ;
                  set   : type_set := upper_case);
    procedure put(to    : string ;
                  item  : enum ;
                  set   : type_set := upper_case);
```

Sortie des caractères constituants le littéral correspondant à `item`*. Le paramètre* `width` *permet de formater la sortie : le littéral est cadré à droite d'une zone comprenant* `width` *caractères. Le paramètre* `set` *permet d'écrire le littéral en majuscule ou en minuscule.*

```
end enumeration_io;
```


E/S des types entiers

```
generic
    type num is range <>;
package integer_io is
    procedure get(file : file_type ;
                  item : out num ;
                  width : field := num'width);
    procedure get(item : out num ;
                  width : field := num'width);
    procedure get(from : string ;
                  item : out num ;
                  last : out natural);
```

Le paramètre `width` *permet d'indiquer le nombre max de caractères à lire.*

```
    procedure put(file  : file_type ;
                  item  : num ;
                  width : field := num'width ;
                  base  : number_base := 10);
    procedure put(item  : num ;
                  width : field := num'width ;
                  base  : number_base := 10);
    procedure put(to    : string ;
                  item  : num ;
                  base  : number_base := 10);
```

Le paramètre `width` *permet le formatage du littéral : cadré à droite d'une zone composée de* `width` *caractères. Le paramètre* `base` *spécifie la base du système de numération.*

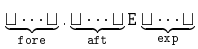
```
end integer_io;
```


E/S des types réels flottants

```
generic
    type num is digits <>;
package float_io is
    procedure get(file  : file_type ;
                  item  : out num ;
                  width : field := 0);
    procedure get(item  : out num ;
                  width : field := 0);
    procedure get(from : string ;
                  item : out num ;
                  last : out natural);

    procedure put(file : file_type ;
                  item : num ;
                  fore : field := 2 ;
                  aft  : field := num'digits-1 ;
                  exp  : field := 3);
    procedure put(item : num ;
                  fore : field := 2 ;
                  aft  : field := num'digits-1 ;
                  exp  : field := 3);
    procedure put(to   : string ;
                  item : num ;
                  aft  : field := num'digits-1 ;
                  exp  : field := 3);
```

Paramètres de formatage :




```
end float_io;
```


Entrées-sorties des types réels fixes

```
generic
    type num is delta <>;
package fixed_io is
    procedure get(file  : file_type ;
                  item  : out num ;
                  width : field := 0);
    procedure get(item  : out num ;
                  width : field := 0);
    procedure get(from : string ;
                  item : out num ;
                  last : out natural);

    procedure put(file : file_type ;
                  item : num ;
                  fore : field := num'fore ;
                  aft  : field := num'aft ;
                  exp  : field := 0);
    procedure put(item : num ;
                  fore : field := num'fore ;
                  aft  : field := num'aft ;
                  exp  : field := 0);
    procedure put(to   : string ;
                  item : num ;
                  aft  : field := num'aft ;
                  exp  : field := 0);
end fixed_io;
```

La compilation séparée

◇ Les motivations

- écrire la définition d'entités et leur déclaration dans des fichiers sources distincts (→ masquage de l'implantation).
- découpage d'un projet au sein d'une équipe de développement
- gestion des différentes versions (archivage des sources dès que le programme semble correct).

◇ Les moyens offerts par Ada

- des unités exploitent des unités existantes :

  la **dépendance** entre unités → clause **with**
- une sous-unité n'est pas définie dans l'unité qui la contient, mais en dehors :

  les **définitions séparées** → clause **separate**

La notion de dépendance ▷

◇ Définition

Si une unité U exploite une unité V qui n'est pas dans sa portée, alors une **clause de dépendance** doit indiquer la dépendance de U vis à vis de V :

```
with V;
définition de U
```

La relation de dépendance est transitive : si une unité U_1 dépend d'une unité U_2, qui elle-même dépend de U_3, alors U_1 dépend de U_3.
Une unité ne peut pas dépendre d'elle-même.

▷ La clause use

ATTENTION : lors de l'étude des paquetages, la clause **with** a été évoquée, dans un sens identique à celui présenté ici (voir page 94). La clause d'utilisation **use** n'a de sens que pour les paquetages.

◇ Élaboration d'une unité dépendante

Si une unité U_1 dépend d'une unité U_2, alors U_1 n'est élaborée qu'**après** qu'U_2 ait été élaborée.

◁ La notion de dépendance

◇ Un exemple

Exemple :

La procédure P *suivante dépend de la fonction* F *et du paquetage* TEXT_IO. *La fonction* F *dépend elle-même d'une fonction* G :

```
function G ( x : float ) return float is
begin
   return x * x ;
end G;

with G;
function F ( x : float ) return float is
begin
   return G(x) + x ;
end G;

with F, TEXT_IO; use TEXT_IO;
procedure P is
   a : float;
   package ESf is new FLOAT_IO(float); use ESf;
begin
   put("Valeur de x : ");
   get(a);
   put("       F(x) = ");
   put(F(a));
   new_line;
end P;
```


Les définitions séparées ▷

◇ Définition

Lors de la définition d'une unité U, il est possible de ne pas écrire la définition d'une sous-unité S **dans** la définition de U, mais **hors** cette définition → la définition de S est alors dite **séparée** de celle de U.

```
unité U is
    ...
    sous unité S is separate;
    ...
end U;

separate( chemin de nommage en partant de U )
sous unité S is
   ...
end S;
```

Les définitions séparées

◇ Un exemple

Exemple :

La procédure `P` *suivante contient une sous-unité fonction* `F` *définie séparément; la fonction* `F` *contient elle-même une sous-unité fonction* `G`*, elle aussi définie séparément:*

```
with TEXT_IO; use TEXT_IO;
procedure P is
   a : float;
   package ESf is new FLOAT_IO(float); use ESf;

   function F (x : float) return float is separate;

begin
   put("Valeur de x : ");
   get(a);
   put("       F(x) = ");
   put(F(a));
   new_line;
end P;

separate(P)
function F ( x : float ) return float is
   function G (x : float) return float is separate;
begin
   return G(x) + x ;
end F;

separate(P.F)
function G ( x : float ) return float is
begin
   return x * x ;
end G;
```


Conventions sur les noms de fichiers

◇ Règles générales

Les conventions suivantes s'appliquent aux fichiers sources ADA :

extension `.ada` : fichier source contenant à la fois des déclarations et des définitions d'unités.

extension `.ads` : fichier source ne contenant que des **déclarations** d'unités (donc aucune définition) → fichier de **spécifications** (d'où le `s` de `.ads`).

extension `.adb` : fichier source contenant les définitions des unités spécifiées dans un fichier `.ads` → d'où le `b` de `.adb`, pour *body*.

Table des matières

Introduction 1
- ADA en quelques mots . . . 2
- Contenu de ce cours . . . 4

Présentation générale 5
- Notions générales sur les unités . . . 7
- Les objets d'une unité . . . 10

Les types 12
- Les sous-types . . . 14
- Les types dérivés . . . 15
- Les types scalaires . . . 16
  - Les types discrets . . . 17
  - Les types réels . . . 25

Les instructions 28
- L'affectation . . . 30
- La conditionnelle - $1^{ère}$ forme . . . 31
- La conditionnelle - $2^{ème}$ forme . . . 33
- Le bloc . . . 35
- Sur la notion d'élaboration . . . 37
- La boucle . . . 38
- Les instructions spéciales . . . 44

Les sous-programmes 47
- Les fonctions . . . 48
- Les procédures . . . 57
- Surcharge . . . 63

Les types composés 64
- Les types tableaux . . . 65
- Les types enregistrements . . . 72
- Les types accès . . . 75
- Les types paramétrés . . . 77
- Les types enregistrements variables . . . 80

Les exceptions 82
- Sémantique des exceptions . . . 83

Les paquetages 85
- Spécification des paquetages . . . 86
- Corps des paquetages . . . 92
- Clause de dépendance . . . 94
- Un exemple de paquetage . . . 95
- Les paquetages prédéfinis . . . 100

Les unités génériques 101
- Les paramètres de généricité . . . 103
- Exemples d'unités génériques . . . 106
- Instanciation . . . 108

Les entrées/sorties 111
- Les fichiers séquentiels . . . 112
  - Le paquetage SEQUENTIAL_IO . . . 113
  - Un exemple d'exploitation . . . 115
- Les fichiers directs . . . 117
  - Le paquetage DIRECT_IO . . . 118
  - Un exemple d'exploitation . . . 121
- Les entrées-sorties textuelles . . . 122
  - Le paquetage TEXT_IO . . . 124
  - Manipulation des fichiers textes . . . 125
  - Canaux standard d'E/S . . . 126
  - Pages, lignes, colonnes . . . 127
  - Pages, lignes, colonnes . . . 128
  - Pages, lignes, colonnes . . . 129
  - E/S des caractères et chaînes . . . 130
  - E/S des types énumérés . . . 132
  - E/S des types entiers . . . 133
  - E/S des types réels flottants . . . 134
  - Entrées-sorties des types réels fixes . . . 135

La compilation séparée 136
- La notion de dépendance . . . 137
- Les définitions séparées . . . 139
- Conventions sur les noms de fichiers . . . 141