# Une

# Introduction

# à ADA

S.Saidi

Département Mathématiques,

Informatique & Systèmes

Ecole Centrale de Lyon
1992-93-94

# *Table des matières*

# Averstissement

Ce polycopié est une introduction au langage ADA qui traite principalement les aspects syntaxiques de ce langage (sans les tâches).

Les pages sont des photocopies des transparents de la partie ADA du cours "Algorithme et Programmation" du cycle A-CNAM qui, utilise en partie, ADA comme langage support.

Nous tenons à remercier particulièrement MM. R.Ogor et R. Rannou de l'ENST Bretagne pour avoir mis à notre disposition un exemplaire du support du cours complet d'ADA dont ils ont la charge à l'ENST. Le présent document s'est largement inspiré de ce dernier.

# Références Bibliographiques

• **ADA** : a Developemental Approach. F. Culwin. Printice Hall. 1992.
• **ADA avec le sourire**. J.M. Bergé, L.O. Donzelle, V. Olive, J. Rouillard. Presse Polytechnique Romande. 1989.
• **ADA Quality and Style**. Software Productivity Consortium. 1991.
• **ADA** : Un apprentissage. M. Gauthier. Dunod Informatique. 1989.
• **Cours Complet d'ADA**. R.Ogor, R. Rannou. ENST Bretagne. 1989.
La partie ADA de ce cours est largement inspiré de ce manuscrit.
Cet ouvrage est actuellement édité sous forme d'un livre.
• **Ingéniérie du logiciel avec ADA**. G. Booch. Inter Editions. 1988.
• **Programmer en ADA**. J. Barnes. Inter Editions. 1988.
• **Reference Manual for the Ada Programming Language**.

D.O.D. 1983.

# Structures de données en ADA

Deux grandes classes de structure de données :

-type à affectation (accès, scalaires, composés) que l'on va étudier
-types sans affectation, ni test d'égalité (tâche et limité privé)

< dessin arbre de présentation >

# Une vue générale des types

a) Types de données scalaires :

- La valeur d'un tel objet n'est pas décomposable.
- Une relation d'ordre total est définie sur les valeurs d'un type scalaire.

Parmi ces types, les entiers et les réels possèdent des propriétés communes. On parle alors des types *numériques*.

## I) Types Discrets (énumérés et entier) :

Chaque valeur possède un rang; il existe une borne inférieure et une borne supérieure sur l'ensemble des valeurs.

❒ I-1) *Types Enumérés* :

•I-1-1) Définition par énumération :

```
type couleur is (blanc, rouge, vert, bleu);
type feu is (rouge, orange, vert);
```

• I-1-2) Caractère :

Le type *Character* est prédéfini et couvre le jeu ASCII

• I-1-3) Booléen :

Le type *Boolean* est prédéfini avec ses deux constantes *true* et *false*

❒ I-2) Type discret Entier :

Le type *Integer* est prédéfini

## II) Types Réels :

*Digits* : réels virgule flottante défini par l'erreur relative maximum (précision).

```
type coefficient is digits 10 range -1.0 .. 1.0;
    -- 10 chiffres signés
```

*Delta* : réels virgule fixe défini par l'erreur absolue maximum (tolérance).

```
type volt is Delta 0.125 range 0.0 .. 255.0;
        -- erreur absolue ±0.125
```

p Le type prédéfini *float* est un type dérivé de *digits* dépendant de l'implantation (digits 6 sur HP).

A tout type scalaire T correspond 2 attributs *T'first* et *T'last* désignant respectivement la première et la dernière valeurs qu'un objet de type T peut prendre.

b) Types de données composés (tableaux et articles).
c) Type accès permettant d'accéder à un objet (les pointeurs)
d) type privés sur lesquels seuls l'opération d'affectation et le test d'égalité sont permises. Ils seront étudiés dans les paquetages.

Divers :

Les types *limité-privés* seront étudiés plus loin.

Le type tâche ne sera pas étudiée.

# Le système de types en ADA

ADA est un langage fortement typé, c'est à dire :

- un objet possède un type
- toute opération sur un type doit appartenir à l'ensemble des opérations connues pour les objets de ce type
- un objet possède une valeur et une seule; celle-ci doit appartenir à l'ensemble des valeurs définies par son type
- aucune conversion implicite de type n'est effectuée sur une valeur.

Les opérations arithmétiques, par exemple, ne sont définies qu'entre entiers de même type, ou qu'entre réels de même type.

| |
|---|
| Tout manquement à ces règles se traduit par une erreur à la compilation ou par le déclenchement d'une exception à l'exécution. |

## Types prédéfinis et types déclarés

- Les types prédéfinis (fournis par l'environnement) :

| **classe** | **mot clé** |
|---|---|
| entier | INTEGER |
| Réel | FLOAT |
| Enuméré | BOOLEAN |
| Enuméré | CHARACTER |
| Tableau | STRING |

• Les constructeurs de types :
  - énumération
  - réel virgule fixe et flottante
  - enregistrement ou article
  - tableaux
  - accès

Il existe d'autres constructeurs tels que
  - type tâche
  - type privé
  - type dérivé

# Equivalence de types

En ADA, l'équivalence de type se fait par nom. Par exemple

```
type couleur is (rouge, jaune, gris, vert, blanc);
type color is (rouge, jaune, gris, vert, blanc);
```

introduisent deux types distincts, bien que textuellement identiques. Ainsi, avec :

```
a : couleur;
b : color;        -- a et b sont des objets de types différents
c,d : couleur;    -- c et d sont du même type que a
e : couleur;      -- ainsi que  e
```

Dans

```
x : ARRAY(0..3) OF integer;
u,v : ARRAY(0..3) OF integer;
```

u et v du même type et différent de x; on ne peut donc pas écrire
x := u;

En ADA, deux objets sont du même type

- s'ils sont associés au même nom de type
- s'ils apparaissent dans une même déclaration d'objets tableaux contraints (voir plus loin).

Exemple :

```
longueur  : integer;
largeur : float;
aire : integer;


aire := longueur * largeur;   -- NON
```

est incorrect car la multiplication est définie sur des entiers ou sur des réels.

Cette notion d'équivalence de type qui permet des contrôles à la compilation exige du programmeur une grande précision. Il doit parfois :

- convertir une valeur d'une type en une valeur d'un autre type;
- utiliser des expressions qualifiées pour lever les ambiguïtés:

## Expressions qualifiées

Une expression qualifiée précise le type d'une valeur. Par exemple :

```
type couleur is (bleu, vert, jaune, noir, blanc, rouge);
type drapeau is (bleu, blanc, rouge);
type table is ARRAY (1..3) OF integer;
```

*couleur'bleu* se lit la valeur bleu du type couleur
*drapeau'bleu* se lit la valeur bleu du type drapeau
*table'(2,3,4)* est une valeur tableau de type table. Il peut y avoir d'autres types tableaux contenant 3 entiers.

## Conversion de type

Il n'y a pas de conversion implicite en ADA. Alors qu'une qualification précise le type d'une valeur, les conversions transforment une valeur d'un type source en une valeur d'une type cible.

Les conversions ne peut se faire que dans les cas suivants :

• Entre types numériques :

```
longueur  : integer;
largeur : float;
aire : integer;

aire := integer(float(longueur) * largeur);
```

• entre type tableaux sous certaines conditions et entre les types dérivés.

# Les sous-types

Certains objets ne prennent qu'une partie des valeurs appartenant à l'ensemble de valeurs d'un type donné. Par contre, ils utilisent l'ensemble des opérations définies pour ce type.

On utilise la notion de sous-types afin d'améliorer les contrôles à la compilation et à l'exécution.

Un sous-type caractérise un sous-ensemble des valeurs d'un type donné dit type de base. Lors de la définition d'un sous-type, on applique une contrainte au type de base.

**contrainte**
**type de base (non contraint) ==========> sous-type**

La contrainte permettant de réduire l'ensemble des valeurs associées au type de base peut être :

- une contrainte d'intervalle pour les types scalaires:

```
SUBTYPE mes_couleurs is couleur RANGE vert..noir;
SUBTYPE majuscule is character RANGE 'A'..'Z';
SUBTYPE petit is integer RANGE 1..10;
```

- une contrainte de précision pour les réels

```
TYPE volt is Delta 0.125 range 0.0 .. 255.0;
SUBTYPE haute_tension is volt DELTA 1.0;      -- contrainte
                                                 -- plus forte
SUBTYPE bad_tension is volt DELTA 0.0125;   --NON
     -- car la contrainte est moins forte que pour volt
```

• une contrainte d'indice de tableau

```
SUBTYPE matrice_carre is matrice(1..5,1..5);
SUBTYPE chaine is STRING(1..26);
```

• une contrainte de discriminant pour les articles

```
SUBTYPE male is personne(sexe=>masculin);
```

p S'il n'y a pas de contrainte => simple renommage :

```
SUBTYPE tes_couleurs is mes_couleurs;
SUBTYPE entier is integer;
```

*La déclaration d'un sous type ne crée pas de nouveau type*.

Il est donc permis de mélanger, dans une expression et sans faire de conversion, des objets appartenant à des sous types différents *d'un même type de base*. Ces mélanges seront contrôlés lors des affectation et passage de paramètres :

```
SUBTYPE petit is integer RANGE -10..10;
SUBTYPE entier is integer;
i : integer;
j : entier;
k : petit;
i := 200;
k := i;          -- c'est légal mais provoquera une exception
                 -- CONSTRAINT_ERROR à l'exécution car on
                 -- sort de l'intervalle de définition.
```

On utilise les sous-types pour déclarer des objets et des paramètres de sous-programmes. Par exemple, pour incrémenter des petits entiers sous contrôle des valeurs :

```
Procédure incr_petit(x : in out petit);
```

# La dérivation

Motivation :

soit les déclarations

```
SUBTYPE mesure is integer;
SUBTYPE poids is mesure RANGE 0..100;
SUBTYPE hauteur is mesure RANGE 0..100;
mon_poids : poids;
ma_hauteur : hauteur;
le_total : mesure;
```

Il est tout à fait légal d'additionner un objet de type poids avec un objet de type hauteur.

```
le_total := ma_hauteur + mon_poids;
```

Pourtant, ceci n'a aucun sens. La dérivation de type d'ADA permet de pallier cet inconvénient.

ADA permet deux sortes de dérivations : avec ou sans contrainte

# La dérivation sans contrainte

A partir de n'importe quel type, appelé type parent, la dérivation permet de définir un nouveau type appelé le type dérivé.

Le type dérivé possède le même ensemble de valeurs que son type parent et conserve les attributs du type parent. Il hérite du type parent un certain nombre d'opérations :

• les opérations prédéfinies sur le type parent

• l'héritage étant transitif, un type dérivé hérite des opération que son type parent lui-même a hérité.

• dans une approche type abstrait, on peut regrouper les définitions et les opérations d'un type A dans un module (package) et tout type dérivé de A héritera de ces opérations.

On distingue des opérations

- communes au type parent et eu type dérivé
- propres au type parent (redéfinitions par le type dérivé )
- propre au type dérivé ( ce que l'on définira après la dérivation)

```
TYPE lumiere is NEW couleur;
TYPE distance is NEW integer;
TYPE mesure is NEW integer;
TYPE poids is NEW mesure;
mon_poids; ton_poids, poids_total : poids;
ma_hauteur : hauteur;
travail : integer;
le_total, une_mesure : mesure;
```

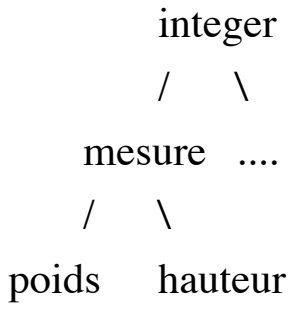
Les opération suivantes sont maintenant illégales :

```
le_total := ma_hauteur+ mon_poids;  -- mélange interdit
mon_poids := ton_poids + ma_hauteur;
```

Par contre, les opérations qui ne mélangent pas les types sont autorisées :

```
le_total := une_mesure * 10;       -- 10 entier universel
poids_total := mon_poids + ton_poids ;
```

Les types *poids* et *hauteur* sont deux types distincts. Cependant sur les types dérivés d'un même parent, il est possible de faire des opérations de conversion explicite. Cette opération est permise entre un type dérivé et son type parent et vice versa.




```
le_total := mesure(ma_hateur) + mesure(mon_poids);
travail := integer(mesure(mon_poids)) *
                        integer(mesure(ma_hauteur));
mon_poids := poids(mesure(ma_hauteur));
```

## *Un exemple plus complet :*

```
Package les_complexes is
    type complexe is private;
    function creer_complexe( reel, imaginaire : float) return
                                                    complexe;
    function "+"(op1,op2 : complexe) return complexe;
    function "*"(op1,op2 : complexe) return complexe;
    -- autres fonctions telles que *, - , /

    private
        -- définition cachée d'un complexe

end les_complexes;
```

On crée ensuite des types dérivés du type complexe qui hériterons des opérations définies sur le type complexe.

```
    type comp_math is new complexe;

    c1 : comp_math := creer_complexe(4.2, 5.0);
    c2 : comp_math := creer_complexe(3.1, 2.0);
    c1 := c1 + c2;          -- l'addition des complexes
```

## La dérivation avec contrainte

On peut ajouter une contrainte lors de la définition d'un type dérivé en faisant suivre le mot clé NEW par une indication de sous-type.

```
TYPE mes_couleurs IS NEW couleur gris..blanc;
TYPE mes_entiers IS NEW integer RANGE 0..100;
```

p Une définition telle que

```
TYPE t2 is new t1 range inf..sup;
```

est équivalent aux déclarations :

```
TYPE t3_anonyme is new type_de_base_de_t1;
TYPE t2 is t3_anonyme RANGE inf..sup;
```

C'est donc le type de base de t1 qui est dérivé, puis contraint. La nouvelle contrainte doit être plus forte où égale à celle qui existait sur t1.

type_de_base_de_t1 == dérivation => t3_anonyme

□ extension □ contrainte

t1 t2

Remarques :

S'il n'y a pas de contrainte, le type dérivé hérite de l'ensemble des valeurs et des opérations.

S'il y a une contrainte, le type dérivé hérite de l'ensemble des opérations et d'un sous-ensemble de valeurs.

*Le type dérivé est un nouveau type, il est totalement distinct du type parent.*

# Les types scalaires

Les types scalaires sont caractérisés par les propriétés suivantes :

- acceptent l'affectation (:=), l'égalité et l'inégalité (=, /=)
- ne sont pas décomposables
- possèdent une relation d'ordre total :
  - opérations de relation : **<, > , <= , >=**
  - test d'appartenance à un intervalle **IN**, **not IN**

- possèdent une borne inférieure et une borne supérieure que l'on peut connaître par les attributs

  **un_scalaire'first** est la borne inf

  **un_scalaire'last** est la borne sup

- des contraintes d'intervalle peuvent être appliquées aux (seuls) types scalaires :

```
SUBTYPE position_pixel IS integer RANGE 1..256;
```

Il y a deux sortes de types scalaires : **Discret et Réel.**

Dans les types discrets, chaque valeur possède un rang, chaque valeur sauf la dernière possède un successeur et chaque valeur, sauf la première possède un prédécesseur.

Dans les types réels, la notion de rang n'existe pas.

Attributs associés aux scalaires : **first, last**

Opérations définies sur les scalaires : **= , /=, < , <= , > , >=, IN, NOT IN**

# Les types discrets :

Il existe deux sortes de types discrets : Entiers et Enumérés.

```
TYPE temperature is range -273.. integer'last;
TYPE couleur is (bleu, blanc, rouge);
```

Les notions de successeur, prédécesseur et rang dans les type discrets se traduisent en ADA par :

**un_discret'succ(x)** où x est une valeur du type de base du sous-type discret un_discret. Le résultat est de type de base de un_discret. Si le successeur de x n'existe pas, l'exception CONSTRAINT_ERROR est levée.

**un_discret'pred(x)** où x est une valeur du type de base du sous-type discret un_discret. Le résultat est de type de base de un_discret. Si le prédécesseur de x n'existe pas, l'exception CONSTRAINT_ERROR est levée.

**un_discret'pos(x)** où x est une valeur du type de base du sous-type discret un_discret. On obtient le rang (un entier universel) de x dans un_discret.

**un_discret'val(x)** où x est un entier universel. Le résultat est une valeur de type un_discret dont le rang est égal à x.

*Exemples :*

```
TYPE couleur is (bleu, blanc, rouge);
integer'succ(4) = 5;
couleur'succ(rouge)   -- non défini, CONSTRAINT_ERROR
couleur'pred(rouge)=blanc
couleur'pos(rouge)=2
```

**couleur'val(1)=blanc**

Les types discrets sont utilisés :

- comme indice de tableau :

**TYPE table is ARRAY (intervalle_discret) OF integer;**

- dans la boucle FOR

**FOR i in intervalle_discret LOOP....**

- dans l'instruction CASE

- Comme discriminant d'un RECORD

❾ D'autres attributs des types discrets :

**un_discret'image(x)** où x est une valeur du type un_discret. Le résultat est une chaîne de caractère du type prédéfini STRING. C'est la forme imprimable de x. Pour un entier, c'est la chaîne des chiffres; pour les énuméré, c'est la chaîne correspondante (BLEU) et pour les caractères, une chaîne composée d'un caractère.

**un_discret'value(x)** où x est une chaîne (STRING) représentant l'image d'une valeur du type un_discret. Le résultat est la valeur discrète dont l'image est x.

**un_discret'width** donne un entier universel la longueur de la plus grande image du type un_discret.

Les opérations prédéfinies sur les types discrets sont les mêmes que pour les scalaires, à savoir **=, /=, < , > , <= , >= , IN, NOT IN**

# Les types numériques

Ce sont les Entiers et les Réels. Seuls les entiers sont discrets.

Opérations prédéfinies : **+, - , * , / ...**

Conversion explicites :

```
float(3);          -- donne 3.0
integer(3.4);      -- donne 3 le plus proche entier
integer(3.5);      -- 4 (ou 3 selon l'implantation)
```

Le compilateur peut déduire automatiquement le type parent d'un type numérique : **TYPE entier is RANGE -10..50;**

Cette déclaration est équivalente à :

```
TYPE entier is NEW integer RANGE -10..50;
```

## *Définition de constante numériques :*

L'expression doit être évaluable à la compilation :

```
cinq : CONSTANT := 5;   -- entier universel déduit
pi : CONSTANT := 3.14159; -- réel universel
deux_pi : CONSTANT := 2.0 * pi;
```

Remarque : dans

```
cinq : CONSTANT := 5;          -- entier universel déduit
five : CONSTANT integer := 5;
```

Les deux déclarations sont différentes. L'une définit un objet (five) de type entier et l'autre introduit une autre façon de nommer 5 (entier

universel cinq). *five* ne peut être utilisé qu'avec des objets de type integer alors que *cinq* est utilisable sans conversion avec d'autres types entiers.

# Types énumératifs

Ce sont des type scalaires discrets.

Le rang du premier élément est 0. La construction d'un type énuméré se fait par énumération des valeurs; celles-ci doivent être distincts les une des autres. Les valeurs sont désignés par des identificateurs ou par des littéraux caractères.

```
Type couleur is (rouge, vert, bleu, marron,.....,noir);
type jour is (lundi,...., dimanche);
type arc_en_ciel is (violet, bleu, vert, rouge, orange);

type hexa is('A','B','C','D','E','F');
type melange is ('A','B', '*', rien);
--p les littéraux caractères sont différents d'un caractère

subtype mes_couleurs is couleur range vert..noir;
subtype week_end is jour samedi..dimanche;
```

Le même littéral d'énumération peut apparaître dans différents types énumérés. Ces littéraux sont dit *surchargés*. Une expression qualifiée par le type de base permet de lever l'ambiguïté dans leur utilisation :

```
couleur'(rouge)   ou    arc_en_ciel'(rouge)
```

*Les opérations associés aux types énumérés :*

Ce sont les opérations définies sur les types scalaires et les types discrets :

```
i : integer;
coul1,coul2,coul3 : couleur;
jour1,jour2 : jour_travail;
planning : ARRAY(jour_travail) OF integer;

coul1 := rouge;       coul2 := coul1;
IF coul2 = rouge THEN...
EXIT WHEN coul3 > bleu;
IF rouge IN coul1 THEN ...
FOR i IN coul2 loop put(couleur'image(i)) ; end loop;

coul3 := couleur'last;            -- il est noir
i := couleur'pos(rouge);      -- i vaut 0
i := arc_en_ciel'pos(rouge);   -- 3
coul1:=couleur'succ(bleu); -- marron
jour1 := jour'val(0);             -- lundi
```

Attributs associés aux types énumérés :

**width, pos, val, succ, pred, image, value, first, last**

Opération prédéfinies sur types énumérés :

**= , =/, < , <= , > , >= , IN , NOT IN**

Les types énumérés comprennent deux types prédéfinis CHARACTER et BOOLEAN.

## Le type Character

Couvre les 128 caractères de la norme ASCII. Les caractères spéciaux sont désignés par une notation pointée :

**ASCII.bel ASCII.bs**

Toutes les opérations applicables aux types énumérés s'appliquent au type caractère :

```
egal, ok : boolean;
un_caracter := 'e';
chiffre := '3';
egal := lettre = '1';
egal := lettre <= chiffre;
ok := chiffre IN '0'..'9';
if lettre NOT IN 'a'..'z' THEN...
lettre := character'succ(lettre);
lettre := character'val(65);          -- 'A'
un_entier := character'pos('A');  -- 65
```

## Le type boolean

Type énuméré prédéfini possédant deux littéraux ordonnés :

*false* et *true* avec *false* < *true*

Outre les opérations sur les types énumérés, les opérations propres aux booléens sont :

**NOT**

**AND , AND THEN**

**OR , OR ELSE**

**XOR** : vrai si les deux opérandes sont différents

Remarques : Malgré les priorités égales de OR, AND et XOR, au lieu d'écrire

**egal OR ok AND trouve** il faut écrire

**egal OR (ok AND trouve)** ou
**(egal OR ok) AND trouve**

Attributs associés aux types booléens :

**width, pos, val, succ, pred, image, value, first, last**

Opération prédéfinies sur les types booléens :

**= , /=, < , <= , > , >= , IN , NOT IN,**
**AND, OR, XOR, NOT , AND THEN, OR ELSE**

# Type Entier

## Type entier prédéfini :

Les entiers prédéfinis sont :

*integer, short_integer, (long_integer).*

## Type entier déclaré

Ce que l'utilisateur déclare :

```
Type pixel is RANGE -256..257;
```

Cette déclaration est une forme abrégée de la dérivation (donc création d'un nouveau type) avec contrainte et avec choix automatique du type parent.

```
x : pixel; y : integer;
y := x;                -- NON, melange de types
y := integer(x);-- OUI, conversion
```

Le type pixel a le même ensemble d'opérations et d'attributs prédéfinis que le type integer.

## Opérations déclarées sur les entiers

a - opération définies sur les scalaires et les discrets

**=, <=, > , >=, IN, NOT IN**

b - les opérations arithmétiques

**+ , - , * , / , MOD , REM , ** , ABS**

La division entière et le reste sont définis par la relation

A = (A/B) * B + (A REM B)

où (A REM B) a le signe de A et une valeur absolue inférieure à B.

La division entière satisfait la relation (-A/B) = -(A/B) = A / (-B)

Le résultat de l'opération modulo est tel que (A MOD B) a le signe de B et une valeur absolue inférieure à celle de B. De plus, le résultat doit satisfaire la relation : il existe un N tel que A=B * N + (A MOD B)

Attributs associés au type entier

**width, pos, val, succ, pred, image, value, first, last**

Opération prédéfinies sur types entiers :

**= , =/, < , <= , > , >= , IN , NOT IN,**

**+ , - , * , / , ** , MOD , REM , ABS**

# Types réels

Le problème des types réel dans un langage de programmation est celui de la représentation finie de suite d'objets susceptibles de prendre un nombre infini de valeurs continue. Il faut donc avoir une présentation suffisamment précise de types infinis par des types finis.

En ADA, on approche les valeurs effectives des réels avec une marge d'erreur relative pour les réels en virgules flottantes et avec une marge d'erreur absolue pour les types en virgule fixe.

Ainsi, une définition de type réel en ADA ne précise pas un type prédéfini mais la précision désirée. Cette précision est soit la précision relative avec l'intervalle de définition des valeurs de types, soit la précision absolue avec l'intervalle de définition des valeurs de type. Le compilateur se charge de choisir le type prédéfini approprié.

## Les types réel à virgule flottante

La précision relative de ces nombres est fixée par le nombre de chiffres pour la mantisse que l'on spécifie après le mot clé DIGITS.

```
TYPE coefficient IS DIGITS 10 range -1.0..1.0;
TYPE reel is DIGITS 8;
TYPE masse is NEW reel DIGITS 10 RANGE 0.0..1.0E10;
SUBTYPE coef_court is coefficient DIGITS 5;
SUBTYPE possibilite is reel RANGE 0.0..1.0;
```

Opérations associés :

Les types réels sont des types scalaires non discrets.

On a les opérateurs : **= , /= , < , <= , > , >= , + , - , / , * , ****

*Exemple :*

```
pi : CONSTANT := 3.14159;
TYPE mon_float is DIGITS 8;
TYPE long is DIGITS 4 range 1.0..200.0;
un_metre : CONSTANTE long := 1.0;
x,y,z : mon_float := 0.0;
l,m,n : long;
y := 0.001266;
l := 23.4;
m := un_metre;
x := un_metre; -- interdit, melange de types
x := l;                -- idem
x := pi;
l := pi;               -- autorisé car pi est un réel universel
l := 300.34;           -- interdit, dépassement de l'intervalle
m := 3.0; n := 4.0;
l := m-n;              -- erreur à l'exécution
```

Les attributs principaux des virgules flottantes :

Si F est un tel type, on a

**F'digits** donne par un entier universel la précision de F

**F'small** réel universel donnant le plus petit nombre positif de F

**F'large** réel universel donnant le plus grand nombre positif de F

**F'epsilon** réel universel donnant la différence entre 1.0 et le premier nombre modèle juste supérieur à 1.

## Les types Réels Fixe

Donnent une précision absolue sur les valeurs effectives. Cette précision est fixée par un nombre de type réel appelé le PAS. Ce PAS est spécifié par l'expression statique située après le mot clé DELTA :

```
TYPE volt is DELTA 0.125 RANGE 0.0..255.0;
SUBTYPE haut_voltage is volt DELTA 1.0;
del : CONSTANT := 1.0/2**(longueur_mot -1);
TYPE fraction is DELTA del RANGE -1.0..1.0-del;
```

L'expression de l'intervalle est obligatoire dans la déclaration du type réel à virgule fixe, elle est optionnelle dans une indication de sous-type.

Opérations prédéfinies :

les mêmes que pour les réels à virgule flottante.

Attributs principaux :

Si F est un tel type, on a

**F'delta** donne le PAS par un réel universel de la précision de F
**F'small** réel universel donnant le plus petit nombre positif de F
**F'large** réel universel donnant le plus grand nombre positif de F

<< tableau des opétaions et des types>>

# Les types composés

Permettent de regrouper plusieurs éléments sous une même structure.

③ le type tableau ARRAY ( ) OF
③ le type enregistrement RECORD .....

## Les tableaux

Un objet tableau est un objet composé d'éléments qui sont tous du même type. Chaque élément est désigné par un indice qui correspond à son rang dans le tableau.

L'indice doit être de type discret (énuméré ou entier). On a

entier : 1,2,..., n --> éléments
caractères : 'a','b','X',...--> éléments
énuméré : rouge,vert,bleu... --> éléments

Exemples :

```
TYPE couleur is (blanc,noir,rouge,vert,bleu...);
TYPE tableau_couleur IS ARRAY
    (couleur RANGE rouge..violet) OF integer;
TYPE tableau_entier IS ARRAY (1..5) OF integer;
TYPE tableau_caractere is ARRAY ('A'..'Z') OF integer;
```

Pour les tableaux à N dimensions, l'ensemble des indices est formé par le produit cartésien de N types (ou sous-types) discrets :

```
TYPE matrice_2 is ARRAY(1..20,1..30) OF integer;
TYPE matrice_3 is ARRAY(1..20,1..30,-4..3) OF integer;
```

Le type des indices doit être discret. Le type des éléments est quelconque y compris un tableau.

```
TYPE matrice_2_2 is ARRAY(1..20,1..30) OF matrice_2 ;
```

Ce qui est différent de :

```
TYPE matrice_4 is ARRAY(1..20,1..30,1..20,1..30) OF integer;
```

Il y a deux types de tableaux :

- avec contrainte : tous les objets d'un tel type ont les mêmes bornes
- sans contrainte : les différents objets d'un tel type peuvent avoir des bornes différentes.

Opérations sur les tableaux :

Les tableaux sont des types avec affectation. Les opérations d'affectation, de test d'égalité et d'inégalité sont donc autorisées sur les tableaux de même type et de même taille.

Les opérations supplémentaires suivantes ne sont définies que sur certains tableaux à une dimension (voir plus loin).

| | | |
|---|---|---|
| si l'élément est | booléen : | NOT, AND, OR, XOR |
| | caractère : | la concaténation & |
| | discret : | <, <= , > , >= |

## Attributs des tableaux

*Ces attributs ne sont applicables qu'à un type tableau contraint ou qu'à un objet de type tableau et non pas à un type non contraint.*

Soit **objet_type** un type tableau contraint ou un objet de type tableau:

| | |
|---|---|
| **objet_type'first** | borne inférieure du premier indice |
| **objet_type'last** | borne supérieure du premier indice |
| **objet_type'range** | intervalle défini par objet_type'first .. objet_type'last |
| **objet_type'length** | nombre de valeurs du premier indice |

Si le tableau est à une dimension, les attributs ci-dessus s'appliquent à celui-ci. Par contre, pour un tableau à plusieurs dimensions, on précisera le rang de l'indice auquel on s'intéresse :

| | |
|---|---|
| **objet_type'first(N)** | borne inférieure du N$^{ième}$ indice |
| **objet_type'last (N)** | borne supérieure du N$^{ième}$ indice |
| **objet_type'range(N)** | intervalle défini par objet_type'first(N) .. objet_type'last(N) |
| **objet_type'length(N)** | nombre de valeurs du N$^{ième}$ indice |

# Les tableaux contraints

Exemples :

```
TYPE couleur is (blanc,noir,rouge,vert,bleu...);
TYPE tableau_couleur IS ARRAY (couleur) OF integer;
```

ou bien

```
TYPE tableau_couleur IS ARRAY
    (couleur RANGE rouge..violet) OF integer;

TYPE tableau_entier IS ARRAY (1..5) OF integer;
TYPE tableau_caractere is ARRAY ('A'..'Z') OF integer;
TYPE jour is (lundi,....., dimanche);
TYPE semaine is ARRAY(jour) boolean;
TYPE ligne is ARRAY(1..max) OF character;
TYPE mot is ARRAY(1..10) OF character;
TYPE phrase is ARRAY(1..5) OF mot;
TYPE texte is ARRAY(1..20) OF phrase ;
TYPE matrice is ARRAY(1..20,1..30) OF integer;
TYPE tableau_dynamique is ARRAY(N-10 .. N+10) OF integer;
```

Remarques :

• la définition de tableau est la seule définition qui peut être faite au moment de la définition d'un objet. Dans ce cas, seule une définition d'un tableau contraint est permise :

```
grille : ARRAY(1..80, 5..100) OF boolean;
melange : ARRAY (couleur RANGE rouge..jaune) OF integer;
page : ARRAY (1..50) OF ligne;
```

• si l'expression définissant les bornes d'un tableau n'est pas une expression *statique*, la taille du type tableau est *dynamique*. Elle dépend de la valeur de l'expression lors de l'élaboration du type (exécution). Une fois évalué, ces bornes restent fixes. Les objets de ce type ont même intervalle d'indice, même taille. (e.g. dans une procédure, avec un type dépendant d'un paramètre inconnue à la compilation ou dans un package générique) :

```
Procedure machine (N : integer) is
        TYPE tabvarie IS ARRAY (1..N) OF integer;
.....
```

• les déclaration de sous-type d'un tableau contraint ne sont pas autorisées:

```
TYPE vecteur is ARRAY(4..26) OF integer;
SUBTYPE interdit is vecteur(8..10);          -- NON
```

Attributs sur les tableaux :

```
grille'first -- 1
grille'last -- 80
grille'first(2)    -- 5
grille'last(2)     -- 100
grille'range       -- 1..80
grille'length(2) -- 96
tableau_couleur'first      -- rouge
tableau_caracter'last      -- 'Z'
```

Opérations d'affectation et d'égalité

```
mat1,mat2 : matrice;
sem1,sem2 : semaine
ents1,ents2 : tableau_entier,

mat1 := mat2;
sem1 := sem2;
if mat1 = mat2 THEN...
ok := ents1 = ents2;
```

Dans ces deux derniers cas, les comparaisons sont faites élément par élément. Les deux objets doivent avoir le même nombre d'éléments.

## Accès aux éléments des tableaux

Pour les tableaux à une dimension, on précise l'indice. Pour un tableau de tableau T, on précise T(i)(j); T(i) étant considéré comme un tableau pouvant recevoir un autre tableau. Pour un tableau à plusieurs dimensions T, on précise T(i,j,...) pour accéder à un élément simple.

### Tableaux à une dimension (vecteurs):

Hors mis les opérations définies sur les tableaux, les opérations **< , <= , > , >= , &** sont définies sur tableaux à une dimension (appelés vecteurs).

p Les opérations < , <= , > , >= ne sont définies que sur les vecteurs dont les éléments sont d'un type discret :

```
IF ents1 <= ents2 THEN.....  -- tableaux de type discret
```

L'opérateur de concaténation & est défini entre deux tableaux de même type et entre un tableau et un élément de tableau :

```
ents1 & ents2
```

représente un tableau de 10 entiers.

Pour un tableau de booléens, on a les opérateurs logiques :

```
sem1 := sem1 AND sem2;      -- tableaux de booléens
```

Accès à une tranche d'un vecteur :

Une tranche est un tableau à une dimension qui désigne une suite de composants consécutifs d'un tableau à une dimension. Remarquons qu'en ADA, un tableau à un indice dont les éléments sont des tableaux est un tableau à une dimension. Une tranche d'un objet tableau est du type de cet objet :

```
une_table_entier(2..4)              -- tranche de 3 entiers
une_table_couleur(vert..violet)  -- tranche de 4 entiers
une_phrase(3..5)                    -- tranche de 3 mots
unmot(4..8)                            -- tranche de 5 caractères

TYPE tab is ARRAY(1..8) OF integer;
TYPE nom is ARRAY(1..8) OF character;
TYPE tab_bool is ARRAY(1..8) OF boolean;
TYPE tables is ARRAY(1..4) OF tab;   -- tableau à une dimension
TYPE matrice is ARRAY(1..8,1..4) OF integer;    -- les tranches
                                -- sont interdites sur plusieurs dimensions
```

```
tab1,tab2 : tab;
nom1,nom2 : nom;
verite : tab_bool;
tables1,tables2 : tables;
mat1 : matrice;

tables1(3..4)    -- tranche de deux tableaux tab.
                                La tranche est du type tables
tables(3)(1..3)  -- tranche de 3 entiers. elle est du type tab
mat1(2..3, 4..6) -- interdit car multi-dimenstionnel
```

On peut effectuer des opérations entre les tranches. Les deux tranches doivent être du même type. Il faut aussi qu'elles contiennent le même nombre d'éléments (l'affectation se faisant élément par élément).

```
tab1(2..4) := tab2(5..7);
tab1(2..4) := tables1(4)(6..8);   -- affectation d'une tranche
                                        -- de 3 entiers
nom1(1..6) := "dupont";  -- une chaine dans une tranche
nom1(0..6) := nom1(1..4);-- NON, pas la même taille
```

Les opérations applicables aux tableaux à une dimension sont aussi applicables aux tranches de tableaux (du même type et de la même taille).

```
IF tab1(2..4) = tab2(5..7) THEN..

nom1(1..8) := "dupont" & nom2(1..2);     -- concaténation
```

Les opérateurs relationnels sur les type des éléments discrets :

```
IF tab1(2..4) <= tables1(4)(6..8) THEN ...
IF nom2(1..4) >nom1(3..6)....
IF nom1(0..6) <=nom1(1..4)...      -- NON pas la même taille
```

Enfin, les opérations logiques lorsque le type des éléments est booléen :

```
verite(1..2)      OR      verite(2..3)
```

Exemple : Palidrome (un mot qui se lit dans les deux sens : ADA, ELLE)

<< palindrome à mettre la >>

# exemple de manipulation de matrice
## 35-1, 2

# Agrégation

L'agrégation est une opération spécifique aux type composés. Elle permet de construire une valeur d'un type composé en donnant les valeurs de chaque composant. Elles sont souvent utilisées pour initialiser les objets composés ou pour leur affecter une valeur soit dans une affectation, soit au retour d'une fonction.

*L'association d'une valeur à un composant doit être faite qu'une seule fois et chaque composant doit recevoir une valeur.*

Les associations de composants peuvent être données soit par leur positions (dans l'ordre du texte pour les enregistrements), soit en nommant les composants choisis, soit en mélangeant les deux méthodes.

Le mot clé OTHERS qui ne peut apparaître qu'en dernier choix permet d'associer une valeur aux éléments restants.

## Les règles d'association : par position, nominative et mixte.

• Par position :

La position de la valeur détermine l'élément auquel elle est affectée (première valeur au premier élément.....)

```
tab1 := (1,3,54,65,32,2,5,0);
tab1(5..8) := (22,56,43,23);
tables1:=((1,45,.....), (.....),(.....),(.....));
tables1 := (tab1,tab2,tab1,(1,2,3,....,8));
tables(2..3) := (tab1,tab2);
```

• Nominative

Chaque élément est nommé (son indice) en lui associant une valeur. L'intérêt de cette notation est de laisser un libre choix dans l'ordre:

```
tab1 := (1..4 => 45, 6..8 =>34, 5 =>21);
tab1(5..8) := (5|8 => 0, 6|7 => 1);
tables1 := (1..8 => 0), (1,2,.....), (1..8 => 1), (45,87,.....));
tables1(2..3) := (2|3 => tab1);
```

Une façon élégante de nommer les éléments restant s'ils doivent avoir la même valeur est d'utiliser OTHERS. Ce choix ne peut apparaître évidemment qu'à la fin.

```
tab1 := (OTHERS => 0);
tab1 := (4|3|2 => 1, OTHERS => 0);
tab1(2..7) := (OTHERS => 1);
```

• Association mixte : on mélange les deux notations mais on donnera la notation positionnelle au début, l'association nominative OTHERS permet de traiter les autres éléments :

```
tab1 := (1,3,45, OTHERS => 0); un_mot := ('b','e', OTHERS => ' ');
tables1 := ((1,3,6,....), 2..4 => (OTHERS => 0));
tab3 : tab := (OTHERS => 0);   -- déclaration et initialisation
tables2 : tables := (OTHERS => tab1);
tables3 : tables := (OTHERS => (OTHERS => 0));
```

Cas de retour d'une fonction

Un exemple de fonction qui met à zéro tous les éléments d'un tableau :

```
Function raz return tab is    begin  return (OTHERS => 0); end raz;
```

Utilisation : tab1 := raz;

exemple page 35-39>>

# Conversion de tableaux

Nous avons vu que pour les opérations prédéfinies sur les tableaux, les objets ou tranches doivent être de même type. Il est possible de convertir une valeur d'un type tableau A en une valeur d'un autre type tableau B à condition :

- que les types tableaux aient même dimension
- que les éléments soient du même type
- que les types indices de même position soient convertibles (entier ou dérivé) :

```
TYPE table1 is ARRAY(1..3) OF integer;
TYPE table2 is ARRAY(1..3) OF integer;
TYPE table3 is ARRAY(1..3) OF float;

tab1 : table1;
tab2 : table2;
tab3 : table3;

tab1 := tab2;              -- NON, types différents
tab1 := table1(tab2);      -- OUI
tab1 = tab3;               -- NON types et éléments différents
tab1 := table1(tab3);      -- NON, types des éléments différents
```

## Les tableaux non contraints

Ces tableaux sont des modèles dans lesquels les domaines de valeurs des indices sont incomplètement spécifies. Les bornes ne sont pas connues lors de la déclaration du type mais définies lors de la déclaration des objets. De ce fait, les différents objets de même type peuvent avoir des bornes différentes.

```
SUBTYPE positif is integer range 1..integer'last;
TYPE vecteur1 is ARRAY(integer range <>) OF float;
TYPE vecteur2 is ARRAY(positif range <>) OF float;
TYPE matrice is ARRAY(integer range <>, integer range <>) OF float;
TYPE vecteur_de_bits is ARRAY(integer range <>) OF boolean;
TYPE romain is ARRAY (natural range <>) OF chiffre_romain;
```

La notation <> est appelée BOITE. Le type ainsi défini est UN TYPE DE BASE. Tous les objets définis à partir de ce type sont de même type (du type de base), même s'ils n'ont pas les mêmes bornes d'indices.

### Les contraintes d'indices :

Une contrainte d'indice est utilisée dans les déclarations de sous-types ou d'objets à partir d'un type tableau non contraint.

Les expressions utilisées dans une contrainte d'indice peuvent être dynamiques. On obtient alors des objets tableaux dynamiques, c'est à dire dont les bornes ne sont connues qu'à l'exécution.

*Toutes déclarations d'objets à partir d'un type de base non contraint doit comporter une contrainte d'indice.*

Exemples :

```
tableau : matrice(1..8,1..8);
rectangle : matrice(1..20, 1..20);
inverse : matrice(1..N, 1..N);       -- N peut être une variable
filtre : vecteur_de_bits(0..31);

SUBTYPE carre is matrice(1..10, 1..10);
SUBTYPE un_vecteur is vecteur(1..10);
SUBTYPE petit_vecteur is un_vecteur(1..5);  -- contrainte plus forte
SUBTYPE ma_matrice is matrice;              -- sans contrainte, deux
                                            -- nomes désignent le même type
```

La contrainte d'indice peut être apportée pour un objet tableau constant par une valeur initiale qui détermine les indices effectifs:

```
mon_vecteur : CONSTANT vecteur2 := (3,4,5,6,7,2,3,5);-- 1..8
le_vecteur : CONSTANT vecteur1 := (3,4,5,6,7,2,3,5);
```

p -- intervalle -32768.. -32761

L'utilisation des tableaux sans contraintes comme paramètres formels permet une plus grande souplesse :

```
TYPE table is ARRAY(integer range <>) OF integer;
SUBTYPE table10 is table(1..10);
t1 : table10;
t2 : table(1000 .. 5000);
t3 : table(50..100);
```

```
Procedure tri(t : in out table) is
    begin
        for i in t'RANGE loop...        -- intervalle effectif de t
    end tri;
```

Utilisations avec appel avec des tableaux de taille différentes :

```
tri(t1);        tri(t2);        tri(t3);
```

Affectation entre les objets d'un tableau non contraint

L'affectation étant autorisée entre objets de même type, elle est permise entre les objets d'un même type tableau non contraint. Cependant, le nombre d'élément de chaque objet doit être identique.

```
petit_vecteur : vecteur2(1..10);
moyen_vecteur : vecteur2(20..40);
grand_vecteur : vecteur2(1..100);   -- trois objets du type vecteur2
mon_vecteur(21..30) := petit_vecteur;
grand_vecteur(1..10) := petit_vecteur;
grand_vecteur(70..90) := moyen_vecteur;

petit_vecteur := grand_vecteur(31..40);
nom := prenom(1..10);
nom_sauce : STRING(1..13) := "sauce tartare";

nom_sauce(7..11) := nom_sauce(4..8);       -- "sauce ce tare";
```

<<ex page 44>

# Les chaînes de caractères

ADA possède le type prédéfini STRING. c'est un tableau non-contraint à une dimension dont les éléments sont des caractères :

```
SUBTYPE POSITIVE is integer RANGE 1..integer'last;
TYPE STRING is ARRAY(positive rang <>) OF character;
```

③ l'indice de début d'un objet de type string est donc forcément ≥1.

Exemples :

```
question : CONSTANT STRING := "combien de caractères?";
```

Dans cet exemple, la contrainte d'indice est apportée par la valeur initiale.

On a *question'first = 1*, *question'last = 22* = nombre de caractères.

Concaténation par &:

Rappelons que "&" est défini entre deux vecteurs (tableau à une dimension) de même type et entre un vecteur et un élément de vecteur.

```
demander_2_fois := CONSTANT STRING := question & question;
nom, mot : STRING(1..10);          -- contrainte explicite d'indice
prenom : STRING(1..15);            -- même type que nom
prenom := "jean claude      ";     -- 15 caractères
mot := nom;
```

Dans le cas des chaînes, la concaténation & s'applique :

- entre deux chaînes : **nom(1..3) & prenom;**
- entre une chaîne et un caractère : **nom(1..3) & 'A'**
- entre deux caractères pour obtenir une chaîne : **'A' & 'B'.**

<< exemple page 215 palindrome>>

**Exemple :**

Remplacer une sous-chaîne par une autre sous-chaîne dans une chaîne :

```
Function remplace(dans, quoi, par : string) return string is
Begin
    IF dans'length < quoi'length THEN return dans;
    ELSIF
        dans(dans'first .. dans'first+quoi'length-1)=quoi THEN
        return    par &
                  remplace(
                      dans(dans'first+quoi'length..dans'last),
                      quoi, par);
    ELSE
        return    dans(dans'first) &
                  remplace(
                      dans(dans'first+1..dans'last),
                      quoi, par);
    End if;
End remplace;
```

Utilisation :

```
    reponse  :=
        remplace("ada est grand, mais ada n'est  pas simple !!",
                 "ada", "ADA") ;
```

ce qui donne :

```
"ADA est grand, mais ADA n'est  pas simple !!"
```

# Les types Enregistrements (Articles)

Un type enregistrement est un produit cartésien de types. Un tel type possède des composants appelés *champs* ou *éléments*. Le regroupement des divers champs sous une même structure se fait pour des raisons logiques. Par exemple, on regroupe les informations concernant un individu (nom, prénom,....).

A la différence des tableaux, les éléments d'un article ne sont pas forcément du même type et l'accès à un élément ne se fait pas par un index, mais en nommant l'identificateur du champ.

Comme pour les types tableaux, on distingue en ADA :

• les types articles contraints : correspondent à la notion mathématique de produit cartésien d'ensembles. Tous les objets d'un même type contraint ont le même ensemble de valeurs.

• les types articles non contraints : les objets d'un même type non contraint peuvent avoir des ensembles de valeurs différents. L'usage d'un tel type correspond à la notion mathématique de l'union de types.

p Objets contraints et non contraints :

Soit un type non contraint T . On distingue deux cas de figures :

• T est un type tableau non contrant => un objet (une variable) d'un tel type a une taille fixe pendant toute sa durée de vie;

• T est un type article non contraint => un objet (une variable) d'un tel type peut, par affectation globale voir sa taille et donc son ensemble

de valeurs changer. Dans ce cas, un type non contraint permet de déclarer des objets non contraints.

Outre leur utilisation comme l'union de types, les types articles non contraints permettent de manipuler des objets qui changent de taille à l'exécution.

## Déclaration d'un type article

Cas de type article contraint :

Une séquence de déclarations de composants. Chaque composant ressemble à une déclaration d'objet :

```
Type la_personne is RECORD
        nom : string(1..20);
        prenom : string(1..15);
        age : natural := 0;
END RECORD;
rene, andre, gerald : la_personne;
```

p Le type article est le seul type sont les éléments peuvent posséder des valeurs initiales par défaut. Ces valeurs seront prises en compte si dans la déclaration de l'objet aucune initialisation n'est spécifiée (cf. le champ age ci-dessus).

```
Type nom_mois is (janvier, fevrier,....., decembre);
Type date is RECORD
        jour : integer range 1..31 := 31;
        mois : nom_mois := decembre;
        an : integer range 1000.. 2999 := 1950;
```

```
END record;
demain : date; -- initialisée au 31 decembre 1950
```

Remarque :

Si l'on ne sait pas quelle information le type article contiendra, on déclare un type article nul. L'intérêt est de pouvoir déclarer des objets de type article et préciser ultérieurement ce que chaque article contiendra :

```
Type une_personne is RECORD
    Null;
END RECORD;
marie_france, pierre : une_personne ;
```

Cas de type article non contraint :

La déclaration des composants est suivie d'une partie variante (voir plus loin).

## Opérations sur les articles

Un type article est un type à affectation (mais ce n'est pas un type scalaire). L'affectation ainsi que les tests d'égalité et d'inégalité sont définis.

```
Type complexe is record
    re : float := 0.0;          -- valeur initiale par défaut
    im : float := 0.0;
END Record;

x,y : complexe;
x:=y;
```

```
If x=y THEN...
rene := andre;
if rene = andre THEN ...
```

p Le type article n'étant pas un type scalaire, on ne peut pas comparer deux objets de ce type par les opérateurs **>** , **>=** , **<** et **<=**.

## Attributs associés aux articles :

Soit **un_objet**, un objet de type article :

**un_objet'constrained** à résultat booléen indique si cet objet est contraint. Si un_objet est un paramètre formel de type non contraint, la réponse dépend de l'objet paramètre effectif en cours.

Attributs divers : Les attributs *address* et *size* s'appliquent aux objets de type article. Les attributs *base* et *size* s'appliquent aux types articles.

## Accès à un composant d'un article :

Cet accès se fait par la notation pointée :

```
demain.jour := 2;
demain . mois := juin;

Type personne is RECORD
    nom : string(1..10);
    prenom : string(1..15);
    age : natural := 0;
    date_de_naissance: date;
END record;
```

```
un_nom : Constant string := “LA_FORET   “;
un_prenom : string(1..15) := “albert            “;
albert, michel : personne;
albert.nom(2) := ‘A’;
albert.nom(4..9) := “ALBERT”;
albert.prenom(4..8) := (4..8 => ‘ ‘);
albert.date_naissance.mois := juillet;
demain.mois := date’succ(aujourd_hui.mois);
albert.nom := “dupont     “;
```

Agrégats dans les articles

Permet de construire une valeur de type article en donnant une valeur à chaque champ. Les règles à respecter sont identiques à celles des tableaux.

1 - association par position :

```
demain := (23, avril, 1988);
michel := (“dupuit     “,”michel      “,30, (10,fevrier,1969));
albert := (un_nom, un_prenom, 0, demain);
x := (30.0, 40.0);              -- x de type complexe
```

2 - association nominative :

```
demain := (an => 1980, jour => 26, mois => juin);
michel := (date_naissance => (20,aout,1970), age => 18,
                nom => “DUCHATEAU “, prenom => “michel        “);
```

3 - association mixte :

```
demain := (23, an => 1988,  mois => juin);
```

```
michel := (un_nom, “michel       “, date_naissance =>hier, age =>0);
```

Le choix OTHERS peut être utilisé comme pour les tableaux mais attention il faut que les champs restants soient tous du même type.

Comme pour les tableaux, les agrégats d’articles peuvent être utilisés pour :

- donner une valeur initiale dans les déclarations d’objets
- fixer la valeur de retour d’une fonction
- donner une valeur à un paramètre effectif de procédures ou de fonctions.

```
marignan : constant date := (14, avril, 1859);
            -- remplacement des valeurs par défauts
une_date : constant dete := (an => 1993, jour => 2, mois => mars);
le_jour_j : date := (13, mai, 1992);

valeur_complexe : complexe := (re => 5.0, im => 6.6);
```

La règle suivante est respectée lors de la construction d’agrégats :

*Une seule valeur doit être donnée à chaque champ; il n’est pas possible d’utiliser le mécanisme par défaut.*

```
hier : date := (9,9);          -- NON, agrégat non complet
```

Exemple d'agrégats comme valeur de paramètre effectif et comme valeur rendue par une fonction : fonction d'addition de deux complexes :

```
Function add_complexe (z1, z2 : complexe) return complexe is
Begin
      return (z1.re + z2.re , z1.im + z2.im);       -- agrégat résultat
END add_complexe;                                        --  de la fonction
```

Appel utilisant un agrégat comme paramètre effectif:

```
z := add_complexe((1.5,2.0) ,compl2);    -- agrégat
```

Remarque : grâce à la surcharge (voir plus loin), il est possible de donner à la fonction ci-dessus le nom "+" :

```
Function "+" (z1, z2 : complexe) return complexe is
Begin
      return (z1.re+z2.re, z1.im + z2.im);
END "+";
```

Appel utilisant la forme infixée :

```
z := compl1 + compl2;
```

<< exemples pages 160, 161>>

## Type article non contraint

On distingue ce type par une partie discriminante qui figure après l'identificateur dans une déclaration de type article.

```
Type sexe is (feminin, masculin);
Type les_personne (le_sexe : sexe) is RECORD
        nom : string(1..20);
        prenom : string(1..15);
        age : natural := 0;
        CASE le_sexe is
                WHEN masculin  => -- caractéristiques d'un homme
                WHEN feminin => -- caractéristiques d'une femme
END RECORD;
```

Le champ discriminant dont la déclaration ressemble à celle d'un champ est accessible par la notation pontée en lecture seule. Le discriminant doit être de type discret.

```
Type chaine( taille : natural := 80) is record
        champ : string(1..taille);
END record;

Type figure is (cercle, carre, rectangle);

type objet (forme : figure ) is record
        -- .......
END record;
```

```
ma_figure : objet(carre);
ta_figure : objet(rectangle);
titre : chaine(5);
nom, prenom : chaine;       -- longueur par défaut de 80
```

Comme pour les types discrets, réel ou tableaux non contraints, il est possible de déclarer des sous-types (contraints):

```
SUBTYPE les_carres is objet(carre);
SUBTYPE chaines_de_taille_2 is chaine(2);
```

Mais attention à ne pas écrire (en donnant l'intervalle des valeurs):

```
SUBTYPE petites_chaines is chaine(1..10);   -- NON
```

*Contrairement aux tableaux, des objets articles ne sont pas nécessairement contraints.*

p Dans les déclarations ci-dessus :

- *ma_figure* est un carre,
- *nom* et *prenom* sont des chaînes qui au départ sont de taille 80. Mais elles pourront changer de taille par affectation globale. Elles ne sont pas contraintes.
- *titre* est une chaîne dont la longueur est fixée à 5 pour toujours.

On en déduit :

• Les objets articles d'un type article contraint sont obligatoirement contraints.

• Les objets articles, d'un type article non contraint dont les discriminants ne possèdent pas de valeurs par défaut, sont obligatoirement contraints.

• Les objets articles, d'un type article non contraint (dont les discriminants possèdent des valeurs par défaut) sont contraints si la valeur du discriminant apparait explicitement dans la déclaration de l'objet (cf. titre) ou s'il est déclaré comme étant d'un sous-type et non contraint si la valeur par défaut est utilisée (cf. nom).

Rôle du discriminant :

Le discriminant doit être de type discret.

- Il permet d'établir une contrainte d'indice sur un composant de type tableau. Ceci nous permet d'avoir des tableaux dont la taille change en les "cachant" dans un objet article.

- Il permet de choisir la valeur pour la partie variante de l'article

- Il permet d'établir une contrainte de discriminant sur un composant de type article.

p La valeur du discriminant est lue par la notation pointée. Il est interdit d'accéder au discriminant en écriture. Un objet article non contraint ne change de valeur de discriminant que par affectation globale sur tous les champs de l'article.

Discriminant comme contrainte d'indice :

Au même titre que les tableaux non contraints tels que :

```
TYPE vecteur is ARRAY(integer range <>) of float;
TYPE matrice is ARRAY(integer range <>, integer range <>) of float;
```

1- On peut mettre des champs d'article comme étant un tableau non contraint :

```
subtype taille_pile is integer range 1..100;
type pile(taille : taille_pile) is record
    le_contenu : vecteur(1..taille);
    le_sommet : taille_pile;
end record;
```

Attention : on peut pas écrire pour *le_sommet*

```
    le_sommet : integer RANGE 1..taille_pile;
```

Car cela ne correspond pas au rôle défini d'un discriminant.

2- On peut mettre des champs d'article comme étant un tableau non contraint :

```
subtype cote_carre is integer range 0..64;
type carree(cote : cote_carre) is record
    le_carre : matrice(1..cote , 1..cote);
end record;
```

3- On peut définir des objets tableaux qui peuvent changer de taille

```
type mat_carree(cote : cote_carre := 16) is record
      le_carre : matrice(1..cote , 1..cote);
end record;
```

p La déclaration en (2) et en (3) semblent similaires, elles sont pourtant très différentes. En (3), la déclaration permet d'avoir des objets non contraints (tableaux) qui seront des tableaux dont la taille peut évoluer alors que (2) ne le permet pas.

**Résumons :**

Pour créer un objet non contraint à partir d'un type tableau non contraint, il faut :

- mettre le type tableau non contraint dans un record ayant comme discriminant la contraint d'indice de ce tableau;
- déclarer l'objet avec la valeur par défaut du discriminant; c.à.d. ne pas donner la valeur du discriminant à la déclaration.
- si l'objet non contraint doit être initialisé, utiliser un agrégat pour faire une affectation globale.

Exemples :

```
s1 : mat_carree;            -- objet article non contraint de taille=16
s2 : mat_carree(8)   -- objet article contraint de taille=8

s3 : mat_carree(3) := (3, (1..3 => (1..3 => 0.0)));
      -- objet article contraint de cote =3; l'agrégat doit contenir
      -- tous les champs y compris le discriminant

s4 : mat_carree := (3, (1..3 => (1..3 => 0.0)));
      -- objet article non contraint de cote =3; la valeur du
      -- discriminant n'est pas donnée dans la déclaration

SUBTYPE st is mat_carree(10);      -- sous-type contraint
s5 : st;                -- objet article contraint
ss1 : carree(7);        -- article contraint; on doit préciser la valeur
                        -- du discriminant car valeur par défaut absente
```

Exemples de modification de la taille d'un objet :

le changement de taille se fait par affectation globale :

```
s1 := s5;               -- nouvelle valeur de s1.cote = 10
s1 := (4, (1..4 => (1..4 => 0.0))) ;        -- s1.cote = 4
s1 := s2;               -- nouvelle valeur de s1.cote = 8
```

p Mais on ne peut pas écrire :

```
s2 := s3;          -- NON, les deux objets sont
                   -- contraints et de taille différentes
s1.cote := 3;      -- NON, accès en écriture au discriminant
```

p Accès en lecture au discriminant

```
IF s1.cote = 4 THEN .....  -- OUI, lecture
```

*Un objet non contraint est donc une sorte de tableau flexible dont la taille peut varier pendant l'exécution du programme.*

Résumons les différents type de tableaux

| TYPE | Tableau contraint | Tableau non contraint | Tableau non contraint et discriminant |
|---|---|---|---|
| OBJET | Taille fixe tous les objets sont identiques | Taille fixe, chaque objet a sa taille | Taille variable, changement par affectation globale |

La possibilité de faire varier la taille permet de résoudre des problèmes où une certaine flexibilité est demandée aux structures de données.

Exemple : La définition classique et prédéfinie de STRING

```
SUBTYPE positive is integer range 1..integer'last;
TYPE STRING is ARRAY(positive range <>) of character;
```

permet de déclarer des chaînes de taille différentes mais fixe pour chacune. Si l'on veut manipuler des chaînes dont la longueur varie, on peut écrire :

```
SUBTYPE natural is integer range 0..integer'last; -- prédéfini
TYPE chaine(longueur : natural := 0) is RECORD
      contenu : string(1..longueur);     -- 0 => chaîne vide
END record;

nom, prenom : chaine;
prenom := (6,"joseph");
nom := (9, "duchateau");

Function "&" (ch1,ch2 : chaine) return chaine is
Begin
      return(ch1.longueur+ch2.longueur,
            ch1.contenu & ch2.contenu);     -- pas de problème de
                        -- récursivité car ce "&" opère sur deux STRINGs
end "&";

nom := prenom & nom;   -- nom.longueur = 15
                              -- nom.contenu'first =1
                              -- nom.contenu'last =15
```

# Les sous programmes

Un problème résolu suivant la méthode de l'analyse descendante est décomposé en sous-problèmes. Cette décomposition se concrétise en associant un sous-programme à chaque sous problème.

③ Un sous-programme correspond à un sous-problème précis apparu lors de l'analyse du problème. Il peut être écrite séparément et au besoin se diviser lui-même en sous-programmes.

③ L'action réalisée par un sous-programme peut être utilisée en différent points du programme. Il suffira d'en demander l'exécution.

En ADA, un sous-programme peut
- être mis au point séparément;
- être utilisé par d'autres programmes;
- être logiquement regroupé avec d'autres sous-programmes ou d'autres entités dans un paquetage.

Deux sortes de sous-programmes (abstraction) :

❒ Procédure : regroupe une série de commandes (instructions); peut avoir des paramètres comme moyen d'échanger des informations avec l'environnement d'appel et de modifier cet environnement.

❒ Fonction : permet de calculer une valeur d'un type quelconque; utilise des paramètres non modifiables.

La déclaration d'un sous-programme peut se faire en deux étapes :
- déclaration de son entête;
- déclaration de son corps.

L'entête constitue l'interface entre le sous-programme et les unités utilisatrices. Elle peut être spécifiée sans que son corps la suit immédiatement (cf. récursivité croisée).

Exemples de déclaration d'entêtes :

```
Procedure affiche(t : table);
Function sup(x,y : float) return float;
```

Suite à une déclaration d'entête, le sous-programme est connu par son nom, il peut être appelé. Le corps sera défini ultérieurement.

La spécification d'une entête (suivie ou non du corps) contient une liste de paramètres formels. Pour chaque paramètre, on spécifie un nom, un type, un mode de passage et éventuellement une valeur par défaut (seulement si mode = IN).

On distingue trois modes de passage de paramètres :

IN : le paramètre agit comme une constante non modifiable. C'est le mode par défaut.

OUT : le paramètre agit comme une variable qui ne peut pas être lue, mais seulement écrite. Il correspond à un résultat.

IN OUT : le paramètre agit comme une variable qui peut être lue et modifiée. Il correspond à une donnée et à un résultat.

p Les paramètres d'une fonction sont tous en mode IN. Ils ne peuvent (donc) pas être modifiés. Le type de la valeur calculée par la fonction est mis après le mot clé RETURN.

Le corps d'une sous-programme est composé de deux parties :

• Partie déclarative : contient les déclarations classiques de type, variables, constantes, exceptions, d'autres sous-programmes....

• Partie impérative : c'est la description de la réalisation de l'algorithme sous forme d'une séquence d'instructions. Cette partie est éventuellement terminée par une section de description des exceptions (voir plus loin).

Lors de la définition du sous-programme, la spécification (l'entête) de celui-ci est reprise intégralement en tête de sous-programme.

```
Procedure echange(x,y : IN OUT integer);     -- spécification de
.....                                          -- l'interface

Procedure echange(x,y : IN OUT integer) is     -- description
temp : integer;
Begin
      temp := x;
      x := y;
      y := temp;
END echange;
```

Exemple de décomposition et écriture des sous-programmes :

<< p. 193,4,5>>

# Portée, Visibilité et Surcharge

La modularité nécessite des règles de portée et de visibilité.

Les règles de portée définissant, pour chaque déclaration, la région dans laquelle la déclaration a un effet. Ceci permet d'améliorer la lisibilité et la maintenabilité des programmes et d'en faciliter les tests.

Il est possible d'encapsuler ou de cacher, dans des unités du langage, des détails non nécessaires à un certain niveau d'abstraction.

Exemple :

```
Procedure a is
X : integer := 5;      _____________
Y : integer := X+3;    ___
Begin                          portée    Portée
.....                       de Y      de X
END a;
```

# Règles de portée et visibilité :

| |
|---|
| Règle d'ordonnancement : Une entité n'est accessible tant qu'elle n'a pas été déclarée. |

Cette règle rend visible :

- un sous-programme par son nom dont l'entête est déclarée
- les paramètres formel d'un sous-programme
- les déclarations dans la partie déclarative d'un sous-programme
- les déclarations englobantes

<< exemple p. 198,199>>

Règle d'accessibilité ou de visibilité : Dans un sous-programme, les entités accessibles sont les entités déclarées dans ce sous-programme (en fonction de la règle d'ordonnancement), et les entités déclarées dans les sous-programmes englobants.

La portée définit tous les points d'un programme où une entité est accessible :

Règle de portée : La portée d'une entité déclarée dans un sous-programme s'étend à ce sous-programme et à tous les sous-programmes qu'il déclare (sous réserve de la règle d'ordonnancement).

Masquage de visibilité :

Il peut arriver que plusieurs entités sous le même identificateur soient visibles en un point du programme. Lequel accède-t-on avec l'identificateur? Comment accéder aux autres?

Une entité est dite directement visible si son nom sans préfixe suffit pour la référencer.

Une entité est dite indirectement visible si son nom doit être préfixé pour la référencer.

Une entité peut ne pas être directement visible si une autre entité du même nom est déclarée dans un sous-programme plus interne. Cette nouvelle entité masque la première.

```
Procedure Invisible is
compteur : float;
    -- compteur réel est directement visible
    procedure Visible is
    compteur : integer;     -- fin de la visibilté du compteur réel
    Begin                          -- compteur entier directement visible
        -- la variable compteur réelle
        -- est masquée par le variable
        -- compteur entière
    END visible;-   - fin de la visibilté du compteur entier

    Begin
    -- compteur réel est directement visible
    END Invisible;
```

p Si la procédure visible doit accéder à la variable compteur réelle, elle doit la référencer par *Invisible.compteur*

Exemples simples de procédures :

Dans un premier temps, les procédures accèdent aux données globales. Les procédures sont ensuite paramétrées.

<< exemples pages 203,204,205,206>>

## La surcharge des sous-programmes

ADA admet, dans certain cas que plusieurs déclarations introduisant le même identificateur soient directement visibles en un point du programme. C'est le phénomène de *surcharge*. Quand un identificateur surchargé est utilisé, le compilateur détermine le bon en fonction des règles de visibilité puis de surcharge. Remarquons que les mêmes règles interviennent pour les littéraux d'énumération.

Pour lever l'ambiguïté, le compilateur utilise :

- le nombre de paramètres effectifs,
- le type et l'ordre des paramètres effectifs,
- le nom des paramètres formels dans le cas d'une notation nominale,
- le type du résultat dans le cas d'une fonction

Exemple : On peut définir une fonction maximum avec des réels, des entiers,... Le compilateur fait appel à la fonction correspondant au type des paramètres :

```
Function maximum (op1,op2 : integer) return integer;
Function maximum (op1,op2 : float) return float;
Function maximum (op1,op2 : matrice) return matrice;
```

Pour une utilisation plus conviviale, on peut surcharger les opérateurs prédéfinis d'ADA :

AND, OR, XOR, NOT

<, <=, > , >=

+ , - , * , / , ** , MOD , REM

Notons que le fait de pouvoir par exemple additionner deux entiers ou deux réels par le même opérateur “+” est du à la surcharge de cet opérateur.

Remarques :

• *La surcharge de l’opérateur “=“ n’est admise que sur des opérandes de type limité privé pour lesquels cet opérateur n’est pas défini (voir plus loin).*

• *L’opérateur “/=“ est toujours déduit de l’opérateur “=“.*

• *La surcharge de l’affectation “:=“ n’est pas possible.*

## Mécanismes d’appel et de retour de sous-programmes

A l’appel :

Il y a passage de la valeur des paramètres effectifs dans les paramètres formels correspondants dont le mode est IN ou IN OUT. Les paramètres en mode OUT ne reçoivent aucune valeur.

Pendant l’exécution :

Les paramètres formels en mode IN sont des constantes, ils ne peuvent être accédés qu’en lecture.

Les paramètres en mode OUT sont des variables; leur valeur n’est pas accessible, on peut seulement leur affecter une valeur.

Les paramètres formels en mode IN OUT sont des variables à part entière, ils sont accessibles en lecture et en écriture.

A la fin de l'exécution :

Il y a passage de la valeur des paramètres formels, dont le mode est OUT ou IN OUT dans les paramètres effectifs correspondant. Les paramères en mode IN n'auront pas été modifiés.

La fin de l'exécution :

La fin de l'exécution d'un sous-programme intervient

• pour une procédure :
- lorsque la dernière instruction du corps a été exécutéé (END)
- lorsque l'instruction RETURN est exécutée
- lorsqu'une exception a été déclenchée

• pour une fonction :
- lorsque l'instruction RETURN est exécutée. Le corps de la fonction doit comporter au moins une instruction RETURN suivie d'une expression donnant un résultat dont le type est celui spécifié dans l'entête. Cette expression est la valeur de la fonction.
- lorsqu'une exception a été déclenchée

<< exemple de la page 208-209>>

# Appel d'un sous-programme

Un appel de sous-programme est composé du nom du sous-programme appelé suivi d'une liste de paramètres effectifs. La correspondance entre les les paramètres effectifs et les paramètres formels peut être faite de deux manières :

- Par la notation nominative : On précise le nom du paramètre formel. Dans ce cas, l'ordre est sans importance :

```
Procedure memorise(element : IN integer; sur : IN OUT tampon);
memorise(element => 342, sur => buffer);
```

- Par la notation positionnelle : Le nom du paramètre formel est absent. C'est l'ordre des paramètres qui permet d'effectuer la correspondance :

```
memorise(342, buffer);
```

- Par la notation mixte : On précise la notation positionnelle, puis nominative:

```
memorise(342, sur => buffer);
```

Remarques :

- l'appel d'une procédure correspond à une commande (instruction), tandis que l'appel d'une fonction rend une valeur qui peut être utilisée comme opérande dans une expression.

- Lors que le nom d'une fonction correspond à un opérateur prédéfini (surcharge), on peut utiliser les paramètres effectifs comme opérandes.

Exemple :

```
Function "+"(x,y : matrice) return matrice;-- "+" surchargé


m := m1 + m2;              -- utilisation sous forme infixée
m := "+"(m1,m2);           -- une autre manière : notation préfixée
```

Valeurs par défaut :

Il est possible d'associer aux paramètres formels en mode IN des valeurs par défaut. L'évaluation de l'expression par défaut se fera à chaque appel.

Exemple :

```
procedure servez_cafe(  marque : string;
                        force_cafe : force := costaud;
                        temp_cafe : temperature := chaud;
                        option_cafe : option := sucre;
                        nombre_cc : quantite := 10);


servez_cafe("hot coffee");          -- prise des options par défaut
servez_cafe("jacques...", nombre_cc => 20); -- café double
servez_cafe("black", force_cafe => leger, option_cafe => lait);
```

Les paramètres formels qui n'ont pas de valeur par défaut doivent avoir un paramètre effectif correspondant à l'appel.

# Récursivité

- Aucune différence majeure avec Pascal , C ou autres langages.
- Exemples de récursivité simple et croisée (Prédéclaration) .

# Les Paquetages

Les modules sont les briques de base dans la construction d'un programme. La notion de module en ADA est supportée par le paquetage. Les paquetages sont compilés séparément et permettent de se créer un environnement riche en les accumulant dans des bibliothèques.

Notions liées au paquetage :

• Encapsulation : Un paquetage est une collection d'entités et de ressources logicielles "encapsulées". Il permet de regrouper dans une même unité de programme des déclarations de constantes, de types, d'objets et de sous-programmes.

• Abstraction : Un paquetage permet de séparer la spécification d'un type (ensemble de valeurs et d'opérations sur ces valeurs) de sa mise en oeuvre (notion de type abstrait).

Comme dans les sous-programmes, un paquetage comprend deux parties :

**- une spécification**

**- un corps**

La spécification constitue l'interface entre unité et environnement. Elle comporte une partie visible contenant les déclarations des objets exportés et une partie cachée (privée) qui sert essentiellement dans le cas des types abstraits (voir plus loin).

Le corps réalise la mise en oeuvre des ressources définies dans la spécification. Elle peut comporter une partie déclaration et éventuellement une séquence d'initialisation du paquetage. Le corps doit

obligatoirement contenir les corps des unités de programmes définies dans la spécification.

## *La spécification de paquetage :*

Exemple :

```
Package les_complexes IS
Type complexe is record
    re , im : float;
END record;
Function "+"(x,y : complexe) return complexe;
    -- entêtes d'autres fonctions telles que "*", "/"..
END les_complexes;
```

## *Utilisation des paquetages :*

**Visibilité à l'extérieur et à l'intérieur du paquetage :**

Les entités déclarées dans la partie visible (spécification) d'un paquetage ont pour portée celle du paquetage. Elles ne sont directement visibles qu'à l'intérieur du paquetage.

A l'extérieur du paquetage, il faut utiliser la notation pointée :

```
WITH les_complexes;
.....
c1,c2 : les_complexes.complexe;
c1 := les_complexes."+"(c1,c2);
```

Pour éviter la notation pointée, on peut soit utiliser le mécanisme de renommage (voir plus loin) soit utiliser la clause USE. Cette clause permet normalement de rendre directement visible les entités exportées.

```
WITH les_complexes; USE les_complexes;
.....
```

```
c1,c2 : complexe;
c1 := c1 + c2;
```

Remarque sur la clause USE :

*Il est impossible avec la clause USE de masquer/surcharger une déclaration directement visible à l'endroit d'apparition de la clause USE; les entités exportées n'ont pas la priorité.*

```
Package probleme IS
    subtype chaine is STRING(1..20);
    Function "&"(x,y : string) return chaine;
END probleme ;

WITH probleme; USE probleme;
procedure plante is
    CH,b : chaine;
begin
    CH := "Ada " & "est grand";
end plante;
```

L'exécution de ce programme provoque une exception CONSTRAINT_ERROR car l'opérateur & est celui qui est prédéfini sur les STRINGs. Il n'y a pas surcharge car les *chaine* est sous-type de *string*. L'affectation de la chaîne "Ada est grand" à CH provoque l'exception car cette chaîne ne fait pas 20 caractères exactement.

Dans l'exemple ci-dessus, on utilise le paquetage les_complexes qui est compilé séparément. Ce type d'usage correspond à un usage ascendante où l'on utilise des briques de base.

Une autre manière est d'appliquer une méthode descendante en définissant le paquetage à l'intérieur de l'unité qui l'utilise. On peut en suite compiler le corps séparément :

```
Procedure utilisateur is
    Package les_complexes IS
        Type complexe is....
        -- entêtes des fonctions du paquetage
    END les_complexes;

    USE les_complexes;
    c1,c2 : complexes;

    PACKAGE BODY les_complexes IS SEPARATE;
    -- le corps sera compilé séparément
Begin
    c1 := c1 + c2;
END utilisateur;
```

Remarquons que sans la clause USE, la fonction "+" n'est pas directement visible et il faut utiliser la notation pontée.

# Visibilité et héritage

Seuls les entités exportées par le paquetage, c'est à dire définies dans sa spécification sont accessibles de l'extérieur.

Il n'y a donc pas d'héritage en ADA : les entités que le paquetage utilise (par une clause WITH) ne sont évidemment pas accessibles à l'extérieur. La seule solution est de reprendre le même contexte par les mêmes clauses WITH. La clause WITH ne s'applique que sur l'unité où elle est présente.

Exemple :

```
WITH les_complexes; use les_complexes;
Package imaginaire IS
.......
End imaginaire;
```

Les paquetages qui utilisent imaginaire n'ont pas accès au paquetage les_complexes. Ils doivent donc importer les_complexes s'ils veulent l'utiliser.

## Le renommage

Le renommage est utilisé pour simplifier les références aux objets lors que ceux-ci ne sont pas directement visibles et alléger les écritures préfixées, ces références Une déclaration de renommage permet de désigner une entité existante sous un nouveau nom sans que la liaison entre un ancien nom et l'entité soit détruite.

Différents usages sont possibles :

• Utiliser un nom plus simple :

```
x : integer renames gestion.enseignant.compte;
```

Ainsi, x désigne l'entité compte du paquetage enseignant interne au paquetage gestion.

• Rendre plus rapide l'accès à un objet :

```
TYPE tab is ARRAY(etudiant) of note;
Procedure calcul(a: tab; i:etudiant) is
    major_promo : note renames a(i);
```

Ici, major_promo dénote l'élément i du tableau a.

• Renommage de sous-programmes, paquetages ... (voir plus loin)

# Le corps du paquetage

**Forme générale :**

```
Package BODY nom IS
-- déclarations éventuelles

-- eventuellemnet        Begin
                              --  suite d'instructions
                              -- Exception       éventuelles
                                    -- traitement des exceptions
END nom;  -- END obligatoire
```

Le corps du paquetage peut être compilé séparément.

Les entités déclarées dans le corps ont leur portée réduite à ce corps et ne sont pas visibles à l'extérieur du paquetage.

L'élaboration d'un corps de paquetage comprend :

- l'élaboration de la partie déclarative. Les objets locaux du corps sont installés. Cette partie comprend les descriptions des sous-programmes définis dans la partie spécification;
- l'exécution des instructions du corps (entre le Begin éventuel et le End du paquetage). Les objets locaux sont initialisés. Ces initialisations survivent (même après la fin de l'exécution du corps). Ils restent accessibles, en particulier via les sous-programmes exportés par le paquetage. (un appel à un de ces sous-programmes de l'extérieur manipulera les données dans leur forme initialisée).

Exemple :

```
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(   re => x.re*y.re - x.im*y.im ,
                  im => x.re*y.im + x.im*y.re);
    END "*";

    -- d'autres fonctions sur les complexes
-- pas de Begin du corps donc pas d'initialisation
END les_complexes;
```

Le corps du paquetage doit obligatoirement contenir les corps de toutes les unités de programmes définies dans la spécification.

## Paquetage et types privés

Ce mécanisme permet d'implanter les types abstraits, c'est à dire une définition de type et des opérations qui portent sur les valeurs du domaine défini par le type sans toute fois laisser voir les détails d'implantation du type et des opérations.

La spécification comporte une partie visible et une partie privée. Dans la partie visible, on trouve les déclarations d'entités exportées :

• une ou plusieurs déclarations de types privés;
• les déclarations des opérations définies sur les objets des types privés (sous forme de spécification d'entête de sous-programme);
• des déclarations d'exceptions correspondant à des cas d'erreurs de manipulation de ces types;
• déclarations de constantes...

Il y a deux sortes de types privés :

- types privés simples :

```
type germe is PRIVATE;
```

- types privés limités

```
type cle is LIMITED PRIVATE;
```

La structure interne d'un type privé est invisible à l'extérieur du paquetage. L'accès à ces objets n'est alors possible qu'à travers les sous-programmes définis dans la partie spécification du paquetage; Ce qui permet une meilleure protection des objets.

Les opérations d'affectation et les tests d'égalité et d'inégalité restent possibles pour les objets d'un type privé (mais pas les autres tests).

La limitation des opérations sur les types privés ne s'applique qu'à l'extérieur du paquetage qui le déclare. Dans le corps même du paquetage, ce type conserve toutes ses opérations.

Exemple :

```
Package les_complexes IS
    Type complexe is PRIVATE;
    Function "+"(x,y : complexe) return complexe;
        -- entêtes d'autres fonctions telles que "*", "/"..

    PRIVATE
        Type complexe is record
            re , im : float;
        END record;
END les_complexes;
```

Ainsi, les utilisateurs du paquetage les_complexes n'ont plus accès à la structure d'un objet de type complexe. Ils ne peuvent donc pas faire référence aux champs *re* et *im* de ce type.

Si l'on doit donner la possibilité d'accès à ces champs, on déclare dans la partie spécification du paquetage les sous-programmes permettant cet accès.

Par exemple, on déclare :

```
Function reelle(nombre:complexe) return float;
    -- rend la valeur de la partie réelle d'un nombre complexe
Function imaginaire(nombre:complexe) return float;
    -- rend la valeur de la partie réelle d'un nombre complexe
Procedure initialiser(re,im : float; nombre : out complexe);
    -- initialise un nombre complexe
```

Les détails d'implantation d'un nombre complexe sont cachés. On peut décider de les implanter autrement; par exemple, par un tableau à deux éléments :

```
Private
    type indice is (re,im);
    type complexe is ARRAY(indice) of float;
```

Remarque : la déclaration de type privé peut comporter une liste de contraintes de discriminants. Le type caché est alors un type article non contraint :

```
Type symbole(size : positive) is Private;
```

Il est possible de déclarer des constantes d'un type privé dans la partie visible (constante différées). La valeur de la constante est inconnue à l'extérieur du paquetage. La déclaration ne comporte alors que le nom et le type de la constante. La partie privée doit déclarer la valeur de la constante :

```
Package protection IS
    Type mot_de_passe is private;
    code : CONSTANT mot_de_passe;
    Function codage(nom : string) return mot_de_passe;
    Function "<"(op1,op2 : mot_de_passe) return mot_de_passe;

PRIVATE
    Type mot_de_passe is range 0..9999;
    code : CONSTANT mot_de_passe := 0;
End protection;
```

Exemple d'utilisation :

```
Use protection;
mon_code : mot_de_passe := code;
```

Il est possible de masquer un opérateur prédéfini en le redéfinissant dans la partie visible. Dans l'exemple précédent, l'opérateur "<" masque celui déclaré sur les entiers.

La surcharge/masquage de l'opérateur "=" n'est possible que pour les types limités privés sur lesquels cette opération est interdite.

## Les paquetages prédéfinis

L'annexe C du manuel de référence fournit la spécification du paquetage STANDARD. Ce paquetage constitue l'environnement de tout programme ADA. Il contient les déclarations des types, constantes, exceptions et opérateurs prédéfinis du langage. ceux-ci sont visibles directement. On les utilisera avec la notation pointée dans le cas de masquage.

## Méthodologie

Sous-programmes et paquetages jouent un rôle fondamental dans la programmation ADA et en particulier la structuration de gros logiciels. Les paquetages permettent de :

- regrouper une collection de déclarations;
- former des bibliothèques d'unité de programmation;
- définir des types abstraits;

Collection de déclarations :

Afin de faciliter les modifications et améliorer la lisibilité des programmes, on regroupe logiquement les entités telles que :

- les constantes d'un même système d'unité;
- l'ensemble des types et des objets globaux d'un programme et leur initialisations.

Le paquetage peut être utilisé pour regrouper des objets communs à plusieurs modules. Cette pratique a cependant le défaut d'encourager le programmeur à utiliser des variables globales.

Regroupement d'unités de programmation

Cet usage correspond à la définition de bibliothèques de sous-programmes. La spécification du paquetage contient alors des déclarations de type, de constantes, de sous-programmes, voir d'exceptions. Le corps du paquetage ne contient alors que les corps des sous-programmes définis dans la partie spécification.

Définition d'objets abstraits :

Une bonne pratique de programmation consiste à n'exporter qu'un type privé par paquetage avec les opérations applicables sur ce type.

On étudie ci-dessous un exemple de paquetage d'un type abstrait ensemble de caractère. Lors de l'étude de la généricité, nous verrons comment définir un ensemble de n'importe quel type.

La définition d'un ensemble est celle des mathématiques. Les valeurs possible d'un ensemble de caractères sont données par tous les caractères possibles.

## Exemple : paquetage ensemble

L'exemple suivant décrit le paquetage ensemble (non générique)

Exemple Ensemble (95)

# La Compilation séparée

Constat : Les environnements de programmation sont de plus en plus riches. La construction d'un système informatique ressemble de plus en plus à un assemblage de pièces d'un jeu de "légo"; c'est à dire, par un assemblage d'unités.

Dans une telle démarche, le concept de "module" est central. Ces modules qui peuvent être "génériques" (modules paramétrés) sont produits par la compilation "séparée" de programmes réalisant des tâches bien définies avec une interface propre et claire.

En ADA, le concept paquetage, bien supérieure au sous-programme, est le moyen de construction de briques de base de grosses applications. Un paquetage permet au programmeur non seulement de regrouper des déclarations de sous-programmes, mais également d'assembler des déclarations de types et d'objets utiles à d'autres parties de son application.

On peut par exemple acheter des paquetages (sous forme source ou binaire), en produire par les membres de l'équipe chargée du développement et les assembler à l'aide de la compilation séparée.

La compilation séparée en ADA supporte deux méthodes de construction de programmes :

•La méthode descendante (sous-problèmes auxquels on fait correspondre des sous-unités)

• La méthode ascendante qui consiste à mettre en bibliothèque des unités de programme dont on a un usage important et varié.

La généricité permet d'élargir le domaine d'utilisation d'une unité. Une unité générique (paramétrée) définit une famille d'unités ne différant que par un certains nombre de caractéristiques (différentes valeurs des paramètres). La définition d'une unité générique procède par généralisation puis, par la création d'un exemplaire spécialisé.

## La compilation séparée

Fait partie intégrante d'ADA. Par sa définition, le langage ADA met en place les mêmes contrôles pour un programme écrit en un seul morceau que pour un programme formé de plusieurs unités compilées séparément.

L'utilité de cette compilation est évidente (et nécessaire) dans les grosses applications sur lesquelles travaillent plusieurs personnes. Une fois définies et compilées les interfaces; le travail peut être réparti.

La compilation séparée permet également de se constituer progressivement un environnement de travail de plus en plus riche lorsque l'on travaille dans un environnement personnalisé.

On peut compiler séparément le corps d'une unité et ainsi d'utiliser dans la construction d'un programme une unité non encore réalisée. Ce qui permet de retarder des choix de mise en oeuvre.

Le découpage en sous-unités permet d'éviter des recompilations inutiles (nécessaires pour une application écrite en un seul morceau).

Le langage ADA distingue deux sortes d'unités de compilation :

❒ Les unités **primaires** :

Ce sont :

- spécification de sous-programme
- spécification de sous-programme générique
- spécification de paquetage
- spécification de paquetage générique
- corps de sous-programme

Ces unités sont les briques de bases dans l'approche ascendante. Il s'agit d'une unité de compilation "importée" (par la clause WITH) réutilisable dans divers contextes.

Exemples :

les unités de la bibliothèque d'entrées-sorties,
les unités de l'allocation / libération, de conversion,
les paquetages personnels...

❒ Les unités **secondaires**

Ce sont les unités que l'on déclare SEPARATE et qui sont détachées de leur contexte de définition et d'utilisation. Une unité secondaire dépend obligatoirement (directement ou non) d'une unité primaire. Elle ne peut pas être utilisée seule.

Une unité secondaire peut être soit un corps d'unité primaire, soit une sous-unité. Ces unités couvrent les corps des sous-programmes, les corps de paquetages et les corps des tâches. Elles permettent de mettre en ouvre l'approche descendante.

## Approche descendante

Dans cette approche, on procède par décomposition de problèmes en sous-problèmes et par raffinement successifs. A un niveau donné de la décomposition, quand on effectue la réalisation d'une unité, on précise seulement la spécification des sous-unités utilisées; la réalisation de ces sous-unités est repoussée à plus tard.

Le corps d'une sous-unité peut donc constituer une unité de compilation (secondaire). La spécification de celle-ci doit apparaître dans l'unité qui l'utilise (unité parente). Le corps est remplacé dans l'unité parente par une **souche** (stub) et le mot clé SEPARATE indique que ce corps est compilé séparément.

Une souche peut remplacer un corps de :

• sous-programme :

```
Procedure placer(tab : in out vecteur; i : integer)
                                    IS SEPARATE;
```

• paquetage :

```
Package BODY fifo     IS SEPARATE;
```

• type tâche :

```
TASK BODY semaphore IS SEPARATE;
```

Ainsi, l'unité parente contenant les souches de sous-unités peut être compilée sans que le corps des sous-unités l'aient été.

L'unité parente peut être elle-même sous-unité d'une autre unité. Lors de la définition du corps d'une sous-unité, on indique :

SEPARATE(chemin_d'accès_à_la_sous-unité)

*Les objets visibles, directement ou non, à l'endroit où apparaît la souche de la sous-unité sont aussi visibles dans le corps de la sous-unité. Dans l'exemple suivant, les variables globales sont toutes visibles. La dépendance entre une sous-unité et l'unité parente induit un ordre de compilation et de recompilation (voir plus loin).*

Le chemin d'accès à une sous-unité part d'un corps d'unité primaire et désigne de façon non ambiguë une sous-unité. Il s'agit d'un chemin dans un arbre. Ce chemin ne contient que des noms d'unités compilées séparément car une souche ne peut apparaître que dans la partie déclarative du corps d'une unité de compilation.

ex p. 105

## L'approche Ascendante : La clause WITH

Cette approche consiste à bâtir un programme à partir d'unités de programme dont la spécification a été préalablement compilée et mise dans une bibliothèque ADA (unités primaires).

Pour utiliser ces unités, on insère dans l'unité utilisatrice, dans la partie appelée *contexte*, des clauses WITH indiquant les unités primaires que l'on veut utiliser. Cette partie contexte précède le texte de l'unité en cours de définition :

```
WITH text_io; Use text_io;
WITH integer_io; Use integer_io;
Package BODY mon_paquetage IS
.....
End mon_paquetage ;
```

Dans cet exemple, on indique que le corps du paquetage *mon_paquetage* utilise l'unité de compilation prédéfinie *text_io*. La clause **Use** permet d'accéder directement, sans passer par la notation pointée, aux objets exportés par *text_io*.

L'effet (la portée) de la partie contexte d'une unité s'étend :
- à l'unité de compilation
- à son corps s'il s'agit d'une spécification
- à ces sous-unités

Placer une clause WITH devant une spécification alors que seul le corps utilise l'unité exportée par la clause WITH peut entraîner des recompilations inutiles. Il est de bonne pratique de ne placer une clause WITH que là où elle est strictement nécessaire : Un corps ou une sous-unité peut posséder une partie contexte.

## Quelques paquetages prédéfinis

- calendar : gestion de temps
- system : caractéristiques dépendant de la machine
- machine_code : programmation langage machine
- unchecked_conversion : conversion de type sans contrôle
- unchecked_deallocation : désallocation mémoire
- sequential_io : entrées/sortie séquentielles
- direct_io : fichiers à accès direct
- text_io : fichiers texte
- io_exceptions : exceptions dues aux entrées/sorties
- low_level_io : entrées/sorties physiques

# Construction d'une application

## Ordre de compilation - Recompilation

Etant donnée les dépendance entre les unités, elles doivent être compilées dans un certain ordre :

- un corps doit être compilé après sa spécification (relation $\mathbb{C}$)
- une sous-unité être compilée après l'unité parente (relation $\mathbb{S}$)
- une unité de compilation (unité primaire, corps ou sous-unité) doit être compilée après les unités de compilation mentionnées dans les clause WITH de sa partie contexte (relation $\mathbb{W}$).

Un programme ADA peut donc être représenté par un arbre de dépendance dont les noeuds sont des unités de compilation et les arcs des relations de dépendance (C, W et S).

dessin p. 109

Le graphe traduit les règles de portée et de visibilité du langage ADA. Il induit un ordre de compilation et un ordre de recompilation. Une modification dans une unité de compilation nécessite la recompilation de toutes les unités qui en dépendent directement ou non.

Dans l'exemple précédent, une modification du corps de Z nécessite aucune recompilation (aucune unité ne dépend de ce corps). Une modification de la spécification de Z nécessite une recompilation du corps de Z ainsi que de la sous-unité P.

Il est important de ne placer les clauses WITH que là où elles sont nécessaires.

```
WITH x;
WITH y;
WITH z;
Package a is
      -- comme l'exemple précédent
End a;
```

Une modification de la spécification de Z dans le programme ci-dessus nécessite une recompilation de la spécification de a, du corps de a et de la sous-unité P (dépendant de le spécification de Z)  bien que seule la sous-unité P utilise l'unité Z.

Quand une sous-unité de compilation n'utilise aucune entité de son unité parente, il peut être intéressant d'en faire une unité primaire de compilation.

Exemple :

```
Procedure application is
    Procédure p(compteur : in out integer) is SEPARATE;
End application;
```

Dans ce cas, une modification de l'unité Application entraîne une recompilation inutile de la sous-unité P.

```
WITH p;
Procedure Application IS
...
End Application;
```

Dans ce cas, la recompilation de l'unité primaire P est évité lors que Application est modifiée.

## Programme Principal - Edition de liens

En ADA, aucun mot clef n'existe pour désigner à la compilation quelle est l'unité de compilation constituant le programme principal. Cette notion n'apparait qu'à l'édition de liens.

On peut désigner un sous-programme d'une unité primaire comme étant le programme principal lors de l'édition de liens. Ainsi, un même sous-programme peut être programme principal dans une application et simple sous-programme interne dans une autre. Remarquons qu'une implantation pourra imposer des restrictions sur las paramètres du sous-programme qui est le programme principal de l'application.

A l'éditions des liens, l'on vérifie que le programme est complet et que tous les corps des sous-unités constituant l'application ont été compilés. L'édition de liens vérifie que toutes les unités de compilation sont à jour, c'est à dire qu'une modification d'une unité A et sa recompilation a bien entraîné la recompilation des unités dépendant de A.

## Ordre d'élaboration

L'élaboration d'un programme ADA débute par l'élaboration des unités de compilation (élaboration des spécifications et des corps) dont a besoin le programme principal. Il s'agit des unités de bibliothèques mentionnées dans les clauses WITH du programme principal, de son corps et de ses sous-unités. D'une manière générale et de façon transitive, il est nécessaire d'élaborer toutes les unités de bibliothèques appartenant à l'arbre de dépendance du programme.

<< dessin p. 113>>

# Les Exceptions

Un programme ne se déroule pas toujours comme prévu, plusieurs cas d'erreurs peuvent venir perturber son bon déroulement : division par zéro, dépassement des bornes d'un tableau, débordement de la pile, fichier inexistant, erreur de type repérée à l'exécution...

On ne peut pas toujours provoquer un arrêt du programme à cause de ces erreurs; par exemple, dans un système embarqué, ceci reviendrait à arrêter le système ( fusée, avion, chaîne de production). Il faut donc être capable de tolérer les pannes. Les exception en ADA constituent un outil pour résoudre ce genre de problèmes.

Une exception en ADA est dite déclenchée. Ce déclenchement peut être effectué par :

- le matériel (division par zéro);
- l'exécutif ADA ( erreur de fichier)
- le programme lui même

Dans le dernier cas, deux possibilités existent :

- les exception prédéfinies qui se déclenchent automatiquement lors que les erreurs qu'elles représentent surviennent;
- les exception définies dans le programme que l'on déclenche explicitement (par l'instruction RAISE). Ce déclenchement ne témoigne pas forcément d'une erreur, mais par fois d'un cas non habituel (exceptionnel).

Une exception peut être traitée localement dans l'unité de programme dont l'exécution a déclenché l'exception. C'est le rôle de *traiteur* (récupérateur) d'exceptions.

Une exception non traité localement est propagée à l'unité appelant. Cette propagation continue tant qu'aucun traiteur n'est rencontré. Elle cause l'arrêt du programme si aucune traitement n'est trouvé. Une exception traité localement peut être propagée afin d'effectuer un traitement au niveau appelant.

Concernant le traitement des exceptions, on trouve les schémas suivants dans la littérature :

• Schéma de détection/correction : on effectue une récupération et le contrôle est repassé à l'endroit ou l'exception a eu lieu.

• Schéma de terminaison : l'unité de programme traite éventuellement l'exception et se termine (écriture d'un message, modification de certains paramètres de sortie);

• Schéma nouvel essai : un nouvel essai du programme par le même algorithme (ou un autre) est tenté. Ce schéma nécessite la restauration du contexte (mise en oeuvre assez lourde).

ADA met par défaut en ouvre le schéma de terminaison. Cependant, on peut réaliser les autres schémas.

## Déclaration des exceptions

Une exception peut être :

- prédéfinie
- fournie par un paquetage prédéfini
- déclarée par l'utilisateur

## Les exceptions prédéfinies

- constraint_error
- numeric_error
- program_error
- storage_error
- tasking_error

Les situations qui peuvent déclencher ces exceptions sont :

CONSTRAINT_ERROR :

- sortie d'un intervalle, d'une contrainte d'index, d'une contrainte de discriminant
- accès à un champ d'article inexistant (record non contraint)
- accès par un pointeur NULL

NUMERIC_ERROR :

- résultat d'une opération prédéfinie hors de l'intervalle numérique de la machine (en particulier division par zéro)

PROGRAM_ERROR :

- en particulier lors qu'un appel à un sous-programme survient alors que le corps de celui-ci n'a pas été élaboré

- STORAGE_EROR :

manque de place lors de l'élaboration des déclarations;
d'une demande d'espace mémoire par un allocateur

- TASKING_ERROR : erreur de tâches.

## Exception fournies par un paquetage

C'est par exemple le cas de l'exception TIME_ERROR du paquetage CALENDAR, ou les exceptions liées aux entrées/sorties sur les fichiers dont les déclarations sont faites dans le paquetage IO_EXCEPTIONS.

## Les exceptions déclarées par l'utilisateur

Forme générale :

identificateurs : EXCEPTION;

Exemples :

```
interblocage : EXCEPTION;
pile_vide, pile_pleine : EXCEPTION;
```

## Traitement des exceptions

Un bloc ou un corps d'unité de programme (tâche, sous-programme, paquetage) peut contenir une partie de traitement d'exception, démarrant par le mot clef EXCEPTION et se terminant par le mot clef END du bloc ou de l'unité. La partie exception peut contenir plusieurs traiteurs qui récupèrent différentes exception propagée éventuellement dans le bloc ou l'unité.

Exemple :

```
BEGIN
.......
EXCEPTION
    WHEN pile_pleine | storage_error =>
        put_line("la pile est pleine");
    WHEN pile_vide =>
        put_line("la pile est vide");

    WHEN constraint_error | data_error =>
        put_line("problème de contrainte");
        RAISE constraint_error;
        -- ici, on redéclenche pour l'unité appelant
    WHEN OTHERS =>
        put_line("une exception non prévue");
        un_param_de_sortie := une_valeur_spéciale;
        -- ici par exemple, on positionne une variable
End unité;
```

Rappelons que certaine de ces exceptions peuvent avoir été déclenchées ou propagées à partir des sous-programmes que notre unité manipule.

Le traitement des exception est séquentiel. Une exception NE peut être présente plusieurs fois dans la partie exception. La clause OTHERS qui ne peut apparaître que dans le dernier récupérateur et comme seul choix, permet de récupérer n'importe quelle exception. Quand aucun traiteur d'exception n'est trouvé pour une exception, celle-ci est propagée à l'unité appelant.

## Déclenchement et Propagation d'une exception

Le déclenchement d'une exception peut être

❒ **explicite** : effectué à l'aide de l'instruction RAISE

Exemple :

```
IF index_pile = 0 THEN RAISE pile_vide;
Elsif index_pile = max_pile THEN RAISE pile_pleine;
End;
```

Il est possible de déclencher explicitement n'importe quelle exception, prédéfinie ou non.

❒ **implicite** (cas d'exception prédéfinie)

Le déclenchement entraîne l'abandon de l'exécution normale du bloc ou de l'unité où a eu lieu l'exception.

On considère les cas de figures suivants :

1- si le bloc ou l'unité comporte un traiteur pour cette exception, l'action associée est exécutée. Il s'agit donc d'un saut au traitement de l'exception. L'unité en cours se termine après ce traitement. L'unité appelant ne sait pas que son appel à l'unité en défaut s'est mal terminé, sauf si dans la partie traitement un paramètre est positionné ou si l'exception est redéclenchée.

2- si le bloc ou l'unité ne comporte pas de traiteur pour cette exception, l'unité en cours se termine à l'instruction qui l'a déclenché. Cette exception est propagée à l'unité appelante avec les mêmes règles. Cette propagation peut donc continuer tant que l'unité qui la recoit n'a pas de traiteur pour cette exception. s'il n'y a aucun récupérateur, deux cas peuvent être considérés :

- la propagation arrive au programme principal et l'exécution du programme s'arrête avec en général un message donnant le nom de l'exception;
- une des unités appelant est une tâche. Il n'y a pas de propagation au-dehors de la tâche. Celle-ci s'arrête et le reste du programme continue de s'exécuter.

Remarque : Le compilateur a parfois la possibilité de détecter qu'une opération entraînera le déclenchement d'une exception. Il l'indique par un message lors de la compilation laissant à l'utilisateur le choix d'exécuter malgré tout le programme ou non.

>> ex p. 121, 121-1>>

Remarques :

• Lors qu'une exception DATA_ERROR survient pendant la lecture d'une valeur, la valeur lue reste disponible et peut être lue sous un autre type à la prochaine lecture. L'instruction SKIP_LINE permet d'ignorer cette valeur.

• Il est possible dans un récupérateur d'exception de modifier les paramètres formels de mode OUT et IN OUT d'un sous-programme ou d'exécuter l'instruction RETURN.

• Après l'appel à une unité qui s'est terminée par une exception, certains paramètres (OUT ou IN OUT) ont pu changer de valeur. Il s'agit souvent des gros types (tableaux, articles). Pour signaler des incohérences éventuelles, on ajoute souvent un paramètre supplémentaire indiquant qu'il y a eu une exception. cette valeur pourra être consultée pour entreprendre des actions appropriés lors des modifications de ces gros types.

```
Procédure P(...;  bien_passe : out boolean) is
Begin
    ....
    bien_passe := true;        -- la procédure s'est bien exécutée
Exception
    When OTHERS =>
        bien_passe := false;       -- y a eu un problème
End P;
```

**Exemple :**

p. 122-1..

# Les structures de données dynamiques

Mêmes principes que ceux vus dans le cas général.

On note cependant que :

• La plupart des langages (e.g. Pascal,C) imposent une gestion personnalisée de la libération des emplacement réservés dans le TAS.

Certaine implantations d'ADA mettent en place une procédure automatique de ramassage des miettes pour récupérer les emplacements qui ne sont plus utilisés.

• En ADA, un premier niveau de déréférenciation est effectué automatiquement. Le symbole ^ de déréférenciation est représenté par le mot clé *all* pour accéder à l'objet pointé dans sa totalité. Hors mis ce cas, les autres accès n'ont pas besoin de déréférencer l'accès.

• En ADA toute déclaration d'une variable de type accès initialise cet objet (le pointeur) à la constante NULL. Cette valeur indique que la variable ne contient pas d'accès sur un objet.

Déclaration d'un type accès :

```
Type acces_entier is ACCESS integer;
a_entier1, a_entier2 : acces_entier;          -- accès à un entier

Type tab is ARRAY(1..8) of integer;
Type acces_tableau is ACCESS tab;          -- accès à un tableau
a_tableau1, a_tableau2 : acces_tableau ;
tab1 : tab;

Type date is RECORD
      jour : integer range 1..31
      mois : nom_mois;
      an : integer range 1000 .. 2999;
End record;

Type acces_date is ACCESS date;
a_demain, _hier : acces_date;          -- accès à un article
demain : date;
```

Création par l'allocateur :

Pour créer un objet de type accès, on utilise l'allocateur NEW. Cet allocateur crée l'objet et donne comme résultat une valeur d'accès qui désigne cet objet. On peut initialiser l'objet de type accès lors de son allocation par une expression qualifiée.

```
a_entier1 := new integer;                -- a_entier1 contient NULL
a_tableau1 := new tab;
a_demain := new date;
a_entier2 := new integer'(45);        -- initialisation à 45
a_tableau2 : new tab'(OTHERS => 0);
a_demain := new date'(jour => 3, mois => fevrier, an => 1988);
```

Accès aux objets créés

L'accès se fait par la notation pointée. L'utilisation du mot clé **all** permet d'accéder globalement à l'objet.

```
a_entier1.all := 20;          -- l'objet accédé par a_entier1 vaut 20
a_hier.all := demain;            -- affectation d'enregistrement
a_tableau1. all := tab1;      -- affectation de tableaux
put(a_entier1.all);           -- écrit 20
put(a_entier2.all);           -- écrit 45
```

❒ Utilisé sans notation pointée ou sans indice, un objet de type accès désigne l'accès sur un objet :

```
a_hier := a_demain;              -- affectation de pointeurs
a_entier1 := a_entier2;          -- accès sur le même objet
```

❒ Utilisée avec la notation pointée, un objet de type accès désigne l'accès à un composant d'un article (déréférenciation):

```
a_demain.jour := 24;
a_demain.mois := juin;
```

❒ Utilisée avec un indice, un objet de type accès désigne l'accès à un élément d'un tableau :

```
a_tableau1(2) := 20;             -- un élément du tableau accédé
a_tableau2(2) := a_tableau1(5);
```

## Exemple

```
WITH text_io, integer_io;
Use text_io, integer_io;
Procedure affectation_acces IS
    type acces_entier is ACCESS integer;
    x,y : acces_entier;
Begin
    x := NEW integer;
    x.ALL := 5;
    y := x;
    put(y.ALL);          -- écrit 5
    y.all := 7;
    put(x.ALL);          -- écrit 7
End affectation_acces;
```

Exemple (pointeur sur pointeur) :

```
with text_io; use text_io;
procedure essptr is
type pt1 is access integer;        -- pointeur sur un entier
type pt2 is access pt1;            -- pointeur sur pointeur sur entier
i1 : pt1;
i2 : pt2;

begin
      i2 := new pt1;
```

```
      i1 := new integer;
```

```
      i1.all := 12;
```

```
      put(integer'image(i1.all));          -- écrit 12
```

```
i2.all := i1;
```

```
i2.all.all := 125;
put(integer'image(i1.all));          -- écrit 125
end essptr ;
```

## Les structures de données récursives

Une structure dont la définition contient une référence à elle-même. Cette définition est forcément un enregistrement.

=> les listes, les arbres,.....

Déclaration en ADA à l'aide d'une déclaration incomplète :

```
Type cellule;   -- déclaration incomplète; ne peut être utilisée
                -- que dans une déclaration de type accès
Type lien IS ACCESS cellule;

Type cellule IS RECORD     -- déclaration complète
    valeur : integer;
    suivant : lien;        -- récursivité dans la déclaration
End record;
```

Une liste simplement chaînée sera une suite de cellules. Une liste comporte toujours un accès sur la première cellule (tête).

**Exemple :** Ecrire un programme qui crée une liste d'entiers. Ces entiers sont lus au clavier et le contenu de la liste est ensuite écrit sur le terminal dans l'ordre inverse.

```
WITH text_io, integer_io;
USE text_io, integer_io;
Procedure liste IS
Type boite;
Type pt_boite is access boite;
Type boite is RECORD
        info : integer;
        svt : pt_boite;
    end record;
p,r : pt_boite := NULL;
i : integer;
Begin
    while not end_of_file loop
        put("donner un entier ");
        get(i); skip_line;
        r:= new boite'(i,p);
        p := r;
    end loop;
    put_line("les valeurs lues sont (dans l'ordre inverse) :");
    r:=p;
    while r /= NULL loop
        put(r.info); r := r.svt;
    end loop;
End liste;
```

<< exemple page 28 >>

## Structures récursives complexes

On peut mettre plusieurs accès dans une cellule. Par exemple, pour construire une liste doublement chaînée, on prévoit un accès sur la cellule suivante et un accès sur la cellule précédente:

```
Type cellule is RECORD
    valeur : integer;
    suivant, precedent : lien;
End record;
```

On peut également introduire un accès sur un fils gauche et un autre sur un fils droit pour produire un arbre binaire :

```
Type noeud IS RECORD
    valeur : integer;
    filsgauche, filsdroit : lien;
End record;
```

```
-- ecrire un programme qui lit une série des chaînes au clavier et les insère
-- dans un ABOH puis affiche le contenu de l'arbre dans l'ordre

WITH text_io; USE text_io;
PROCEDURE arbre IS
TYPE noeud;
TYPE lien IS access noeud;
TYPE noeud IS record
        info : string(1..10);
        gauche,droit : lien;
END record;
arbre : lien := NULL;
ch : string(1..10);
l : integer;

PROCEDURE insere(e : string; arbre : in out lien) IS
BEGIN
        IF arbre = NULL THEN
                arbre := new noeud'(e,null,null);
         ELSIF arbre.info > e THEN insere(e,arbre.gauche);
         ELSE insere(e,arbre.droit);
        END IF;
END insere;
```

```
PROCEDURE affiche(arbre : lien) IS
BEGIN
        IF arbre /= NULL THEN  affiche(arbre.gauche);
                                        put(arbre.info);
                                        affiche(arbre.droit);
  END IF;
END affiche;


BEGIN
        LOOP
                ch := (others => ' ');
                put("donner une chaine (< 10 caracteres) ");
                get_line(ch,l);
                insere(ch,arbre);
        END LOOP;


        EXCEPTION
                when END_ERROR => affiche(arbre);
                when OTHERS =>          put_line("probleme de lecture ");
                                        affiche(arbre);
END arbre;
```

Exemple : paquetage de gestion d'un arbre binaire :

<< exemple p. 184,185>>

## Type accès, tableaux non contraints ou articles à discriminants

Le type accédé peut être n'importe quel type. Il peut y avoir des interdépendances entre plusieurs types accédés.

<< exemple page 31 plus ses commentaires >>

Remarques :

• *Un objet créé par un allocateur est toujours contraint.*

On ne peut donc pas changer les valeurs des discriminants de cet objet.

• Un objet accès peut être contraint ou non contraint. S'il est contraint (contraintes d'indices ou discriminants données à la déclaration), il désigne toujours des objets avec les mêmes valeurs de discriminants ou les mêmes contraintes d'indices. S'il n'est pas contraint, il pourra de manière successive, désigner des objets (contraints) dont les valeurs de discriminants ou de contraintes d'indices sont différentes.

<< ex p. 32 >>

<< exemple page 33>>

## Exercice du Crible d'Eratosthene :

Il s'agit de générer la liste des nombres premiers de l'intervalle 1..N

Deux étapes :

1- génération d'une liste contenant 1..N (procédure Genere)

2- élimination des nombre non-premiers de cette liste (procédure Elimine)

```
Procedure Genere(N: entier; P: sortie lien) =
R : lien;
Debut
    P := nil;
    pour i bas 1..N faire
        nouveau(R); R^.info := i; R^.svt := P; P:= R;
    Fin pour;
fin Genere;

Procedure Elimine(P : lien ) = -- élimination des nombre non-premiers
    Procedure Elimine_multiple(X: entier ;P : sortie lien ) =
            -- élimination des multiples de X dans P
    Debut
        si P /= Nil alors
            si (P^.info mod X = 0) alors
                    P := P^.svt; elimine_multiples(X,P);
            sinon Elimine_multiples(X,P^.svt);
            Fin si;
        Fin si;
    Fin Elimine_multiple;
```

```
Debut      -- Debut Elimine
      si P /= Nil alors
                  elimine_multiples(p^.info,p^.svt);
                  elimine(p^.svt);
      fin si;
fin Elimine;
```

Appel : Genere(N,Liste);
Affiche(Liste); -- affichage du contenu de Liste
Elimine(p^.svt); -- le premier élément est 1, on le laisse
Affiche(Liste);

## Exemple : occurrences avec les listes

On dispose d'un fichier texte contenant des mots. Les mots peuvent être séparés par un ou plusieurs espaces, un point, une virgule ou le retour chariot....

On veut obtenir le nombre d'occurrences de chaque mot de ce texte. Pour cela, on utilisera une liste de boites. Chaque boite contient un mot et le nombre de fois où celui-ci apparait dans le texte.

**Le principe :**

```
ouvrir le fichier F
Liste <- nil
Tant que non fdf(F)
$    lire un mot M
     si M <> mot_vide alors
          si M est dans Liste alors faire +1 sur nb_occ de M
          sinon     insérer M dans Liste
                    mettre nb_occ de M à 1
$
Fermer F
Editer Liste;
```

La lecture des mots :

Pour simplifier, on lit le fichier F à la volée (caractère par caractère) et on construit un mot. Ce qui permet d'utiliser les types chaînes et ensembles Pascal.

 Discuter des avantages et des inconvénients de maintenir la liste triée.

# La Généricité

La généricité permet d'élargir le contexte d'utilisation d'une unité de programme. Elle permet de détacher une sous-unité de compilation définie lors d'une approche descendante de la programmation, de son unité parente et devenir une unité de bibliothèque à part entière.

La généricité permet de définir des familles paramétrées d'unités de programmes, les unités d'une même famille ne différant que par un certain nombre de caractéristiques décrites à l'aide de paramètres formels génériques. La création d'une unité de programme à partir d'une unité générique se fait par **instanciation** (création d'un exemplaire spécifique à partir d'un moule). Lors de cette instanciation on associe des paramètres effectifs aux paramètres formels génériques.

La généricité peut s'appliquer (unité pouvant être générique) :

- aux packages
- aux abstractions (procédures et fonctions)

Les paramètres génériques peuvent être :

- des types
- des constantes
- des objets (variables)
- des abstractions (procédures et fonctions)

## La généricité simple : type en paramètre

Considérons la procédure d'échange de deux objets de type entier.

```
PROCEDURE echange_entier(premier, second : IN OUT integer) IS
tampon : integer ;
Begin
    tampon : = premier ;
    premier : = second ;
    second : = tampon ;
End echange_entier ;
```

La seule opération effectuée sur les objets manipulés par cette procédure est l'affectation. Il est donc *a priori* possible d'échanger des objets du "**type à affectation**" (tous les types sauf **privé limité et tâche**).

Si nous voulons réaliser l'échange avec un type différent, par exemple des réels, nous sommes obligés de réécrire complètement une autre procédure :

```
PROCEDURE echange_reel(premier, second : IN OUT float) IS    tampon :
float ;
Begin
    tampon : = premier ;
    premier : = second ;
    second : = tampon ;
End echange_reel;
```

Pour chaque type d'élément à échanger, nous devons réécrire une procédure. Pourtant, seul le type des objets à échanger diffère d'une procédure à l'autre.

La généricité nous permet d'écrire un moule à partir duquel on peut créer des procédures spécifiques à chaque type (sans avoir à réécrire ces procédures).

Dans le cas de l'échange, seul le type diffère entre les différentes procédures : le type constitue un paramètre générique. Ceci peut être comparé aux paramètres des procédures et des fonctions qui permettent à ces sous-programmes de travailler avec des objets différents (mais de même type).

Les paramètres génériques sont déclarés entre le mot clé **GENERIC** et la spécification de la procédure.

## Déclaration de la procédure générique

```
GENERIC
    type element is private ;
Procedure echange_tout (premier, second : in out element);
```

**IS PRIVATE** signifie ici que le paramètre effectif qui sera associé à **"element"** pourra être de tout type sur lequel l'affectation et la comparaison sont définis (type à affectation)

Le corps de *echange_tout* est le même que celui de *echange_entier*.

```
PROCEDURE echange_tout(premier, second : IN OUT element) IS  tampon :
element;
Begin
    tampon : = premier ; premier : = second ;
    second : = tampon ;
End echange_tout ;
```

Grâce à la procédure générique ci-dessus, il est possible de créer, en les spécialisant par un type précis, de nouvelles procédures. Cette opération de création s'appelle l'**instanciation** de procédure générique.

## Instanciation de procédures

```
Procédure echange_caractere IS NEW
        echange_tout(element => character) ;
Procédure echange_integer IS NEW
        echange_tout(element =>  integer);
Procédure echange_float IS NEW echange_tout(float) ;
```

## Mise en oeuvre de la généricité

La généricité s'applique à deux unités de programmes :

- les sous-programmes (procédures et fonctions),
- les paquetages.

*==> Il n'y a pas de généricité possible avec les tâches.*

Considérons le paquetage de gestion de pile permettant de dépiler et d'empiler des entiers.

```
PACKAGE gestion_de_piles_d_entiers IS
    Type pile is private;
    Procedure empiler (la_pile : in out pile ; l_element : integer) ;
    Procedure depiler (la_pile : in out pile ; l_element : out integer);
    Function longueur (la_pile : pile) return natural ;

PRIVATE
    - implantation du type pile non spécifiée
End gestion_de_piles_d_entiers ;
```

Nous allons rendre ce paquetage générique, car la notion de pile est indépendante du type des éléments qu'elle contient. Le type des éléments à empiler constitue donc le paramètre générique.

```
GENERIC
    Type element is PRIVATE;        -- paramètre générique
PACKAGE gestion_de_toutes_piles IS
    Type pile is PRIVATE;
    Procedure empiler (la_pile : IN OUT pile ; l_element : element) ;
    Procedure depiler (la_pile : IN OUT pile;
                                  l_element : OUT element);
    Function longueur (la_pile : pile) return natural ;
PRIVATE
    - implantation du type pile non spécifiée
End gestion_de_toutes_piles ;
```

*Element* est ici un type paramètre formel générique du paquetage *gestion_de_toutes_piles*. A l'intérieur de la spécification et du corps du paquetage générique, on peut utiliser *element* en tant que type.

On indique, par **IS PRIVATE**, qu'on utilise l'affectation (et/ou l'égalité) à l'intérieur du corps. C'est le cas des procédures *empiler* et *dépiler*.

La création d'exemplaires spécialisés se fait par instanciations :

```
PACKAGE gestion_piles_d_entiers IS NEW
        gestion_de_piles(integer) ;
PACKAGE gestion_piles_de_flottants IS NEW
        gestion_de_piles(element  => float) ;
```

entier *et* float *sont ici les paramètres effectifs correspondant à* element (*l'égalité et l'affectation sont autorisés sur les entiers et les flottants)*.

On dispose ainsi de deux paquetages spécialisés fournissant deux types "pile" différents :

```
pile_entier : gestion_piles_d_entiers.pile;
pile_reel : gestion_piles_de_flottants.pile;
```

On dispose maintenant de deux objets qui peuvent être utilisés. Pour alléger les notations, on utilise la clause **USE**, qui rend directement visible l'intérieur des spécifications des paquetages :

```
USE gestion_pile_d_entiers ;
USE gestion_pile_de_flottants ;
    ...
empiler (pile_entier, 3) ;
empiler (pile_reel, 4.5) ;
```

Remarques :

1- Une fois une unité générique spécifiée, on ne peut que :

- lui associer un corps,
- instancier des exemplaires.

2- Un paquetage générique peut constituer une unité de compilation à part entière. Il peut donc servir dans le contexte d'autres paquetages ou d'autres sous-programmes (clause **WITH**). Cependant, il n'est pas possible d'appliquer la clause **USE** sur un paquetage générique, seules ses instanciations peuvent être utilisées avec la clause **USE**.

3- Seule l'utilisation du type **RECORD** avec discriminant permet d'avoir une pile qui peut contenir à la fois un entier et un réel.

**Exercice** : Réutliser la procédure générique *echange_tout* et le paquetage générique *gestion_de_toutes_piles* (dont les définitions sont rappelées ci-dessous) pour écrire un paquetage générique qui offre comme type un vecteur d'éléments, comme objet une pile d'éléments, (élément est le paramètre générique) et comme service des procédures de :

- transfert de la pile vers un vecteur et vice versa,
- inversion dans un vecteur des éléments d'indice faible par ceux d'indice élevé.

```
GENERIC
    type element is private ;
Procedure echange_tout (premier, second : in out element);
PROCEDURE echange_tout(premier, second : IN OUT element) IS  tampon :
element;
Begin
    tampon : = premier ; premier : = second ;  second : = tampon ;
End echange_tout ;

GENERIC
    Type element is PRIVATE;
PACKAGE gestion_de_toutes_piles IS
    Type pile is private;
    Procedure empiler (la_pile : IN OUT pile ; l_element : element) ;
    Procedure depiler (la_pile : IN OUT pile;
                                    l_element : OUT element);
    Function longueur (la_pile : pile) return natural ;
PRIVATE
    - implantation du type pile non spécifiée
```

```
End gestion_de_toutes_piles ;
```

<< exemple page 128>>

## Vers un exemple plus complexe

Soit la procédure de TRI suivante :

```
Type table_entier is ARRAY (1 .. 50) OF integer ;
Procedure tri_entier (tab_a_trier : IN table_entier ;
                                    tab_trie : OUT table_entier) ;
```

Cette procédure permet de trier un tableau d'entiers. Si on désire trier un tableau de caractère ou de réels, il faut écrire d'autres procédures ou rendre cette procédure générique.

```
GENERIC
        Type table is private ;      -- paramètre dit générique
Procedure tri_tout (tab_a_trier : in table ; tab_trie : out table) ;
        -- cette procédure décrit le tri de tableaux de type table
        -- le type n'est pas connu pour l'instant
```

Cette procédure peut être instanciée en lui passant comme paramètre effectif des tableaux à une dimension :

Exemples d'instanciation :

```
        Type table_entier is ARRAY (1 .. 50) OF integer ;
        Procedure tri_entier IS NEW tri_tout(table_entier) ;

        Type table_caractere is ARRAY (1 .. 100) OF character ;
        Procedure tri_caractere IS NEW  tri_tout(table_caractere) ;
```

A ce niveau, on voit apparaître un certain nombre de problèmes :

• Le paramètre précisé lors de l'instanciation doit ici être un type tableau. Pour trier ce tableau, l'algorithme de tri utilise des attributs propres au type tableau. Il faut donc spécifier dans la description de **tri_tout** que seuls les paramètres de type tableau sont acceptés lors de l'instanciation. Un paramètre TYPE générique doit donc, dans certains cas, préciser les **contraintes** que doivent satisfaire les types fournis en paramètres effectifs lors de l'instanciation.

• Les éléments de ce tableau doivent être comparables. S'il s'agit d'un tableau d'articles il faut indiquer ce que veut dire comparer deux articles (article n'est pas un type scalaire, donc pas de relation d'ordre).

Il y a deux solutions à ce problème. On peut soit contraindre les éléments du type tableau à des éléments comparables, soit donner la manière avec laquelle on doit comparer les éléments. Dans ce dernier cas on fournit en paramètre générique la fonction de comparaison (voir plus loin).

Les mécanismes de généricité complet mis en oeuvre en ADA permettent de résoudre ces différents problèmes.

Une unité générique comporte comme toute unité de programme une spécification et, éventuellement, un corps qui peuvent être compilés séparément.

## Spécification générique

Une spécification d'unité générique commence par le mot réservé **GENERIC** précédant la liste des paramètres formels génériques. Seuls les sous-programmes et les paquetages peuvent être génériques.

Les paramètres formels génériques peuvent être de quatre sortes :

- des paramètres **valeurs** (Caractéristique de dimensionnement),
- des paramètres **objets** (Liaison d'une unité à un objet global),
- des paramètres **types** (Tris d'entiers - Tris de nombres flottants),
- des paramètres **sous-programmes** (Tris croissants - Tris décroissants)

Exemple :

```
GENERIC
    type pixel is range <>;        -- entier
    type abcisse is range <>;      -- entier
    type ordonnee is range <>; -- entier
    type image is ARRAY (abcisse, ordonnee) OF pixel ;
PACKAGE logiciel_image is
    -- Partie visible et partie privée du paquetage
End logiciel_image ;
```

L'exemple ci-dessus comporte un paquetage générique possédant quatre paramètres *pixel*, *abcisse*, *ordonnee* et *image*. Le paquetage fournit des logiciels de traitement d'images de taille quelconque pour

lesquelles les pixels n'ont pas leur domaine de valeurs fixé une fois pour toute.

## Instanciation

L'unité de programme générique définit un moule ou "template". A partir de ce moule, il est possible de créer par instanciation des exemplaires d'unité de programme.

Ainsi le paquetage logiciel_image permet de créer deux paquetages *logiciel1* et *logiciel2 :*

```
type petit IS range 0..255 ;
type long IS RANGE 0..511 ;
type petit_pixel  IS range 0..127 ;
type gros_pixel  IS range 0..255 ;
type image1 IS ARRAY (petit, petit) OF gros_pixel;
type image2 IS ARRAY (long, long) OF  petit_pixel ;

PACKAGE logiciel1 IS NEW logiciel_image
                (petit, petit, gros_pixel, image1) ;

PACKAGE logiciel2 IS NEW logiciel_image
                (long, long, petit_pixel, image2) ;
```

Comme pour les paramètres de sous-programmes, il est possible d'associer des valeurs par défaut aux paramètres formels. Les trois notations *positionnelles*, *nominales* et *mixtes* sont utilisables pour effectuer la correspondance entre les paramètres effectifs et les paramètres formels.

## Les paramètres génériques

Nous avons précisé la nécessité d'avoir des paramètres types avec contrainte et des paramètres **sous-programmes**. Le langage ADA permet également de passer des paramètres **valeurs** ou **objets**.

## Paramètres types

Le langage ADA permet d'imposer des contraintes sur les paramètres types effectifs. Déjà l'indication **IS PRIVATE** impose une contrainte sur le type qui peut être passé en paramètre effectif ; il doit être un type à affectation (ne peut pas être une tâche).

Les différentes façons d'imposer une contrainte sont données ci-contre :

1) **TYPE** aucune_contrainte **IS LIMITED PRIVATE** ;
   *-- Tout type autorisé; aucune opération prédéfinie n'est exigée,*
   *-- pas même l'affectation ou l'égalité*

2) **TYPE** un_type **IS PRIVATE** ;
   *-- N'importe quel type pour lequel l'affectation*
   *-- et la comparaison égalité-inégalité est permise*

3) **TYPE** un_pointeur **IS ACCESS** un_type_quelconque ;
   *-- N'importe quel type accès permettant*
   *-- de pointer sur des objets de type* **un_type_quelconque**

4) **TYPE** un_discret **IS** (<>) ;

*-- N'importe quel type discret (entier ou énuméré)*

5) **TYPE** un_entier **IS RANGE** <> ; *-- N'importe quel type entier*

6) **TYPE** un_flottant **IS DIGITS** <>;*-- N'importe quel type réel flottant*

7) **TYPE** un_fixe **IS DELTA** <>; *-- N'importe quel type réel fixe*

8) **TYPE** tableau_contraint **IS ARRAY** (un_type_indice) **OF** element ;

*-- N'importe quel type tableau dont le type indice est*
*-- **un_type_indice** et le type des éléments **element***

9) **TYPE** tableau_non_contraint **IS ARRAY**
(un_type_indice **RANGE <>**) OF element ;
*-- N'importe quel type tableau dont le type indice est un sous-*
*-- type de* **un_type_indice** *et le type des éléments* **element**

La méthode de programmation consiste donc à passer du particulier au général **en regardant quelles sont les opérations prédéfinies et les attributs utilisés dans l'unité de programme.**

Il y a dans ADA une classification des types selon le critère de généralité. A chaque classe de types correspond un ensemble d'opérations prédéfinies et d'attributs. Nous montrons ces classes en utilisant la même numérotation pour les classes que pour les contraintes.

<< dessin arbre des types p. 135>>

On remarque que certaines classes ne sont pas numérotées. Il n'existe donc pas de règle pour fixer une contrainte sur cette classe. Ainsi, on ne peut pas demander en ADA à ce que les paramètres effectifs soient des scalaires, des réels (fixe et flottant) ou des types composés.

Avec l'indication **limited private**, On indique qu'on n'utilise aucune opération ou attribut prédéfini. Les opération dont on a besoin devront être explicitement précisées (voir plus loin).

Exemple : déclaration de la procédure générique qui trie des tableaux dont les éléments sont de type discret :

```
GENERIC
        Type indice is (<>);      -- discret : entier ou énuméré
        Type element is (<>);  -- discret : entier ou énuméré
        Type table is ARRAY (indice) OF element;
Procedure tri_discret (tab_a_trier : in table; tab_trie : Out table) ;
```

Instanciation :

```
Type l_indice is integer range 1..50;
Type tableau_entier is ARRAY (l_indice) OF integer;
Procedure tri_entier IS NEW
            tri_discret(l_indice, integer, table_entier);

Type tableau_caracteres is ARRAY (l_indice) OF character;
Procedure tri_caracteres IS NEW
            tri_discret(l_indice, character, table_caracteres);
```

Remarque :

On peut pas aller jusqu'au bout de ce raisonnement : bien qu'il y ait des opérations communes à tous les types numériques (entiers, réels), on ne peut pas indiquer une contrainte numérique et définir par exemple une fonction somme qui calcule la somme des éléments d'un tableau dont les éléments sont numériques; on ne peut donc pas écrire :

```
GENERIC
    Type un_numerique is NUMERIC;  -- NON, pas en ADA
    .........
```

Pour ce faire, il faudra définir trois unités génériques, chacune paramétrée par un des types numériques.

```
GENERIC
    Type un_numerique is RANGE <> ;        -- les entiers

GENERIC
    Type un_numerique is DELTA <>;      -- les virgules fixes

GENERIC
    Type un_numerique is DIGITS <>;     -- les virgules flottantes
```

De même, on ne peut pas paramétrer par un type scalaire (discret ou réel) et il nous faudra trois unités génériques (pour discret, fixe et flottant).

Il est donc impossible de généraliser la procédure *tri_tout* pour traiter des tableaux d'entiers ou de réels.

La seule solution est de déclarer cet élément de n'importe quel type, c'est à dire *IS PRIVATE*. On pourra ainsi instancier la procédure générique avec des paramètres éléments de type non scalaire (accès ou

composé). Ceux-ci n'étant pas comparables, il sera nécessaire de préciser en paramètre générique la fonction de comparaison.

## Les paramètres sous-programmes

Dans la déclaration suivante, on indique que n'importe quel type privé convient (PRIVATE). On dispose de l'opération d'affectation(:=) et de comparaison (=, /=). Par contre, pour faire des comparaisons telles que (<=, >=, ..), on précise une fonction *compare* en paramètre :

```
GENERIC
        Type element is PRIVATE;
        WITH Function compare (op1,op2: element) return boolean;
        Type indice is (<>);      -- discret
        Type tableau is ARRAY (indice) OF element;
Procedure tri_tout (tab : IN OUT tableau);
```

Utilisation :

```
Type nationalite is (breton, catalan, basque, berrichon);
Type tab1 is ARRAY (nationalite) OF integer;

PROCEDURE tri_croissant IS NEW
            tri_tout(integer, "<=", nationalite, tab1);

PROCEDURE tri_decroissant IS NEW
            tri_tout(integer, ">=", nationalite, tab1);
```

## Paramètres sous-programmes par défaut

On associe des paramètres par défauts aux sous-programmes. Ces paramètres seront pris si l'on ne précise pas le nom du sous-programme lors de l'instanciation :

**IS <>** Par défaut, un sous-programme du même nom que le paramètres formel

**IS *nom*** Par défaut, un sous-programme de nom *nom*

```
GENERIC
    Type element is PRIVATE;
    WITH Function "<=" (op1,op2: element) return boolean is <>;
        -- si ce paramètre n'est pas précisé, on prend "<="
    Type indice is (<>);     -- type discret
    Type tableau is ARRAY (indice) OF element;
Procedure tri_tout (tab : IN OUT tableau);
```

Utilisation :

```
Type nationalite is (breton, catalan, basque, berrichon);
Type tab_entier is ARRAY (nationalite) OF integer;
Type tab_reel is ARRAY (nationalite) OF float;

PROCEDURE tri_entier IS NEW
    tri_tout(integer, nationalite, tab_entier );
    -- Par défaut, "<=" prédéfini sur les entiers est pris

PROCEDURE tri_réel IS NEW tri_tout(float, nationalite, tab_reel );
    -- Par défaut, "<=" prédéfini sur les réels est pris
```

Si l'on veut dans certains cas fixer soi-même l'opération d'égalité, et dans d'autres cas, prendre par défaut l'égalité prédéfinie :

```
GENERIC
    ......
    WITH Function egal(op1,op2 : element) return boolean IS "=";
    .....
```

Le mécanisme de passage de sous-programme est utilisé pour généraliser l'utilisation des sous-programmes et "les passer en paramètre d'un autre sous-programme" comme dans certains langages.

Dans l'exemple suivant, la fonction *integrale* calcule l'intégrale du fonction *f* entre deux valeurs a et b. la fonction à intégrer constitue un paramètre générique :

```
GENERIC
    WITH FUNCTION f(x : float) return float;
FUNCTION integrale (a,b : float; nb_pas : integer) Return float ;

FUNCTION integrale (a,b : float; nb_pas : integer) Return float is
Begin
    ......
    y := a;
    x := f(y);        -- utilisation de la fonction passée
    .........                --en paramètre
    Return(z);
End integrale ;
```

Une instanciation :

```
Function cos(d:float) return float;      -- sera définie plus loin
Function integer_cos IS NEW integral(f=> cos);
```

Une utilisation :

```
integer_cos(a => 3.2, b=> 4.1, nb_pas => 1000);
```

## Les paramètres valeurs

Un paramètre par valeur constitue une constante à l'intérieur de l'unité générique. On peut préciser une valeur par défaut pour un tel paramètre :

```
GENERIC
    ligne : IN integer := 24;
    colonne : IN integer := 80;
PACKAGE terminal is ...
```

Exemples d'instanciation :

```
PACKAGE minitel IS NEW terminal(24,40);
PACKAGE imprimante IS NEW
    terminal(lignes => 66, colonne => 132);
PACKAGE console IS NEW terminal;     -- par défaut 24 et 80
```

Remarque : on peut réaliser le même effet avec les tableaux dynamiques, tableaux non contraints ou articles non contraints.

Un cas d'utilisation : Certaines implantations n'autorisent pas que la procédure principale ait des paramètres. Pour ce faire, il est possible de compiler une instance de procédure générique :

```
GENERIC
    nb_phi : positive := 4;
Procedure table_de_phi ;      -- procédure générique

-- instanciation
Procedure table_de_2_phi is NEW table_de_phi(2);
```

*table_de_2_phi* peut servir de programme principal pour toute implantation.

## Paramètres Objets :

Le paramètre effectif doit être une variable; une variable par défaut peut être associée à la déclaration de paramètre formel. On utilise cette méthode pour détacher de son environnement une variable globale dont le passage en paramètre à une procédure est couteuse.

```
GENERIC
    var_globale : IN OUT un_type;
Procedure travail;
```

Instanciation :

```
Procedure travail1 is NEW travail(globale1);
```

## Généricité et récursivité

A l'extérieur de l'unité générique, le nom d'une fonction générique ne peut être utilisée qu'à instancier un exemplaire. Par contre, à l'intérieur de l'unité générique, on peut utiliser ce nom et donc pour la récursivité :

```
GENERIC
    Type type_entier is Range <>;
    Function Factorielle(d : type_entier) return type_entier;
    Function Factorielle(d : type_entier) return type_entier is
    Begin
        -- rappel récursif de la fonction Factorielle
    End Factorielle;
```

## Généricité et compilation séparée

Contrairement aux autre unités non-génériques, le corps d'une unité générique doit être compilé avant toute instanciation. C'est la spécification plus le corps de l'unité générique que constituent un moule.

Une instanciation d'unité générique peut constituer une unité de compilation. Dans l'exemple ci-dessous, le paquetage *entier_io* est crée par instanciation du paquetage générique *integer_io* interne au paquetage *text_io* :

```
WITH text_io; use text_io;
Package entier_io is NEW integer_io(integer);
```

```
-- création du paquetage entier_io pour faire des entrées sorties d'entiers.
```

# Exemple: package générique implantant un ensemble

```
GENERIC
        TYPE element IS private;
        WITH function egal(e1,e2:element) return boolean IS "=";
        WITH procedure afficher(x:element);
PACKAGE ensemble_generique IS
        TYPE ensemble IS PRIVATE;
        ens_vide : constant ensemble;
        function dans(x:element; e:ensemble) return boolean;
        function "*"(e1,e2: ensemble) return ensemble;

        function "+"(e1,e2: ensemble) return ensemble;
        function "+"(e:ensemble; x:element) return ensemble;
        function "+"(x:element;e:ensemble) return ensemble;

        procedure affiche_ensemble(e:ensemble);

PRIVATE     -- implantation de l'ensemble sous forme de liste
        TYPE ele;
        TYPE ensemble IS access ele;
        TYPE ele IS record
                info : element;
                svt : ensemble;
        END record;
        ens_vide : constant ensemble := NULL;
END ensemble_generique;
```

```
PACKAGE BODY ensemble_generique IS

function dans(x:element; e:ensemble) return boolean IS

BEGIN

        if e=ens_vide then return false;

        else return ((e.info = x) or else dans(x,e.svt));

        END if;

END dans;


function "*"(e1,e2:ensemble) return ensemble IS

temp : ensemble;

BEGIN

         if e1=ens_vide then return ens_vide;

        elsif dans(e1.info,e2) then return (e1.info + e1.svt * e2));

        else return(e1.svt * e2);

        END if;

END "*";


function "+"(e1,e2:ensemble) return ensemble IS

BEGIN

        if e1=ens_vide then return e2;

        elsif dans(e1.info,e2) then return (e1.svt + e2);

        else return (e1.info + ( e1.svt + e2));

        END if;

END "+";
```

```
function "+"(e:ensemble;x:element) return ensemble IS
temp : ensemble;
BEGIN
        temp := NEW ele'(x,e);
        return temp;
END "+";

function "+"(x:element;e:ensemble) return ensemble IS
BEGIN
        return (e+x);
END "+";

procedure affiche_ensemble(e:ensemble) IS
BEGIN
         if e /= ens_vide then  afficher(e.info); affiche_ensemble(e.svt); END if;
        END affiche_ensemble;
END ensemble_generique;
```

# Exemples d'utilisation

```
WITH text_io;
WITH ensemble_generique;
procedure test_ensemble_entier IS
procedure aff(e:integer);
PACKAGE mon_ens IS NEW ensemble_generique(element => integer,afficher => aff);
use mon_ens;
e1,e2,e3,e4 : ensemble:=ens_vide;

procedure aff(e:integer) IS
BEGIN  text_io.put(integer'image(e));
END aff;

BEGIN
        e1 := e1 +1 ; affiche_ensemble(e1);text_io.NEW_line;
        e2 := 10 + e2; affiche_ensemble(e2);text_io.NEW_line;
        e3 := e1 + e2; affiche_ensemble(e3);text_io.NEW_line;
        e4 := e1 * e2; affiche_ensemble(e4);text_io.NEW_line;
END test_ensemble_entier;
```

```
WITH text_io;
WITH ensemble_generique;
procedure test_ensemble_chaine IS
procedure aff(e:string);
subTYPE chaine IS string(1..6);
PACKAGE mon_ens IS NEW ensemble_generique(element => chaine,afficher => aff);
use mon_ens;
e1,e2,e3,e4 : ensemble:=ens_vide;
procedure aff(e:string) IS
BEGIN
        text_io.put(e(e'first..e'last));
END aff;
BEGIN
        e1 := e1 + "ada   ";
        e1 := e1 + "est   "; affiche_ensemble(e1);text_io.NEW_line;
        e2 := "genial" + e2; affiche_ensemble(e2);text_io.NEW_line;
        e3 := e1 + e2; affiche_ensemble(e3);text_io.NEW_line;
        e4 := e1 * e2; affiche_ensemble(e4);text_io.NEW_line;
        IF dans("ada   ",e1) then text_io.put_line("vrai"); else text_io.put_line("faux");
        END if;
END test_ensemble_chaine;
```

# Entrées-Sorties en ADA

C'est la façon par laquelle l'ordinateur communique avec l'extérieur.

Le fichier est un support de lecture/écriture sans que l'on ait à se préoccuper de sa gestion physique (magnétique)

Les informations sur les fichiers sont sous deux formes :

- BINAIRE : le fichier n'est pas directement lisible;
- TEXTE : la forme reconnue par les imprimantes et les terminaux. Cette forme (caractères ASCII) est lisible.

## Organisations des fichiers en ADA

• Séquentielle : comme sur une bande magnétique où, pour atteindre une chanson, il faut partir du début et passer par toutes celles qui précèdent.

Pour lire l'élément N d'un fichier, il faut lire les N-1 éléments.

Les fichiers textes sont des fichiers séquentiels.

•A accès direct : Les éléments sont directement accessibles par leur index. Comme sur un disque, on peut choisir la chanson à écouter.

ADA fournit 4 paquetages prédéfinis :

- SEQUENTIAL_IO : pour les fichiers séquentiels
- DIRECT_IO : pour les fichiers à accès direct
- TEXT_IO : pour les fichiers textes
- IO_EXCEPTIONS : liste des exceptions liées aux entrées/sorties.

# Fichiers Textes

Cas simple : Lecture à partir du clavier et écriture sur l'écran.

Pour pouvoir manipuler les fichiers textes dans une unité de compilation, il faut faire précéder celle-ci de :

```
With text_io;
Use texte_io;   -- pour éviter la notation pointée
```

Pour lire et écrire des entiers ainsi que les réels, on peut procéder de deux manières :

♣ créer les instances suivantes de paquetages *integer_io* et *float_io* :

```
With text_io;
Package integer_io IS NEW text_io.integer_io(integer);
With text_io;
Package float_io IS NEW text_io.float_io(float);
```

La compilation de cette unité insère les paquetages *integer_io* et *float_io* dans la librairie. On peut ensuite les utiliser par :

```
With integer_io,float_io; Use integer_io,float_io;
```

 Pour une meilleure adaptation des opérations d'entrées/sorties aux types que l'on déclare (de même que pour les entrées sorties sur les types énumérés), on peut créer des instances des paquetages génériques ci-dessus dans chaque programme :

```
With text_io;
Procedure P IS
Type mon_entier IS ....
Package mon_entier_io IS NEW text_io.integer_io(mon_entier );
```

## Structure d'un fichier texte en ADA

- C'est une suite de pages ;
- Chaque page est formée d'un suite de lignes ;
- Une ligne est formée d'une suite de caractères.

Les pages d'un fichier, les lignes d'une page et les caractères d'une ligne sont numérotées à partir de 1. Le numéro d'un caractère dans une ligne représente sa colonne.

On peut savoir si l'on est à la fin de fichier, de page ou de ligne. Lors que l'on est à la fin du fichier, on est à la fin de ligne et de page:

*end_of_file* implique *end_of_page* AND *end_of_line*

On peut sauter à la page ou à la ligne suivante :

- en entrée : *skip_line* et *skip_page*
- en sortie : *new_line* et *new_page*

## La spécification (paquetage TEXT_IO)

**Type COUNT IS range 0 .. INTEGER'LAST;**

**Subtype POSITIVE_COUNT IS COUNT range 1 .. COUNT'LAST;**

❑ **Function end_of_file return boolean;**

***-- True si la fin du fichier d'entrée est atteinte***

❑ **Function end_of_page return boolean;**

***-- True si la fin de page ou la fin du fichier d'entrée est atteinte***

❑ **Function end_of_line return boolean;**

***-- True si la fin de ligne ou la fin du fichier d'entrée est atteinte***

- **Procedure new_page; -- saut de page en sortie**
- **Procedure skip_page; -- saut de page en entrée**
- **Procedure new_line (spacing : IN positive_count :=1);**
  **-- saut de *spacing* ligne en sortie**
- **Procedure skip_line (spacing : IN positive_count :=1);**
  **-- saut de *spacing* ligne en entrée**

Il est possible de connaître le numéro de la page, de la ligne ou de la colonne courante. De même on peut se positionner sur une ligne ou une colonne :

- **Procedure set_col(to : IN positive_count);**

***-- en sortie, se positionner en colonne to. Pas de retour en arrière***

- **Procedure set_line(to : IN positive_count);**

***-- en sortie, se positionner sur la ligne to sur la page courante (si***
***-- la ligne n'a pas été atteinte).***
***-- En entrée, permet de sauter des lignes.***

- **Function col Return positive_count;**

***-- rend le numéro de la colonne courante sur le fichier de sortie***

- **Function line Return positive_count;**
  ***-- rend le numéro de la ligne courante sur le fichier de sortie***
- **Function page Return positive_count;**
  ***-- rend le numéro de la page courante sur le fichier de sortie***

# Entrées-Sorties de chaînes de caractères

- **Procedure get(item : OUT string);**

**-- lit autant de caractères que la taille de *item*.**

- **Procedure put(item : IN string);**

**-- écrit les caractères de *item* dans la ligne courante**

- **Procedure get_line(item : OUT string; last : OUT natural);**

**-- lecture avec passage à la ligne suivante (*skip_line* implicite).**

**-- *last* est la position du dernier caractère lu.**

- **Procedure put_line(item : IN string);**

**-- écriture avec passage à la ligne suivante (*new_line* implicite).**

# Exemple

```
With text_io; Use text_io;
Procedure lit_ecrit IS
        ch : string(1..20);
        i : integer;
Begin
        put_line("donner une chaîne "); get(ch);
        put_line("le contenu de la chaîne est " & ch);
        put_line("donner une autre chaîne "); get_line(ch,i);
        put_line("la chaîne réellement lue " & ch(1..i));
End lit_ecrit;
```

## Entrées-Sorties d'entiers et de réels

On suppose que les paquetages *integer_io* et *float_io* sont importés par une clause WITH.

▲ Lecture : Par la procédure GET.

Cette procédure saute les caractères blancs, fin de ligne et de page jusqu'à trouver un signe (+,-) ou un chiffre. Elle lit alors le littéral entier ou réel. La lecture s'arrête dès qu'un caractère qui ne fait pas partie d'un nombre est rencontré.

▲ Ecriture : Par la procédure PUT.

Cette procédure imprime une valeur numérique.

Exemple : Supposons qu'une ligne contient *bbb2.3x* -- b = blanc

```
i,j:integer; c,d:character;
get(i);          -- lit 2
get(c);          -- lit '.'
get(j);          -- lit 3
get(d);          -- lit 'x'
```

Avec la même ligne on peut avoir :

```
x : float; d : character;
get(x);          -- lit 2.3
get(d);          -- lit 'x'
```

Exemples d'impressions :

```
put(3);                    ==> bbbbbbbbbb3
put(0);                    ==> bbbbbbbbbb0
put(INTEGER'first);        ==> -2147483648
```

Exemple :

Lire une série d'entiers (100 au plus) et les imprimer à raison de 8 entiers par ligne.

<< exemple page 7>>

## Impression d'un nombre flottant

Un réel est imprimé sous le format : FORE.AFT E EXP

FORE : la partie entière (de longueur 2 par défaut)

AFT : la partie fractionnaire (de longueur *float'digits-1* par défaut)

EXP : la partie exposant (de longueur 3 par défaut)

Ce format permet d'imprimer un nombre réel avec float'digits chiffres décimaux significatifs. La partie entière contient un chiffre (normalisation)

Exemples (pour *float'digits* = 6):

```
put(123.5);          ==> b1.2300E+02
put(-0.356E7);  ==> -3.560006+06
```

## Les exception liées aux entrées-sorties

Les exceptions suivantes sont disponibles dans le paquetage *text_io* (à l'origine, elles sont déclarées dans le paquetage *io_exceptions* et renommées dans *text_io*).

END_ERROR : déclenchée lorsqu'une fin de fichier est atteinte lors d'une opération GET, SKIP_LINE, SKIP_PAGE,...

DATA_ERROR : déclenchée lors d'une opération de lecture de nombre entier ou réel (GET), lorsque les caractères lus ne correspondent pas à un littéral d'entier ou de réel. Le caractère fautif reste le caractère courant.

Exemple : Si une ligne contient *bbb.3* et on exécute *get(i)* avec i de type integer ou réel, l'exception DATA_ERROR est déclenchée et le caractère courant sera le '.' que l'on pourra consommer par une lecture de caractères.

# Gestion des fichiers

=> trois sortes de fichiers prédéfinis en ADA :

- A accès séquentiel (paquetage SEQUENTIAL_IO)
- A accès direct (paquetage DIRECT_IO)
- Fichier texte (paquetage TEXT_IO)

Les deux premiers sont génériques et le paramètre générique correspond au type des éléments du fichier :

```
With io_exceptions;
GENERIC
    type element_type IS PRIVATE;
Package sequential_io IS  .....
```

Pour pouvoir utiliser un fichier séquentiel (ou à accès direct), il faut inclure le paquetage *sequential_io* par une clause WITH puis créer un exemplaire de ce paquetage générique en fournissant le type d'élément du fichier :

Exemple (pour un fichier séquentiel) :

```
With sequential_io;
Procedure P IS
    Type etudiant IS RECORD
        nom, prenom : string(1..20);
        age : positive;
    End Record;
Package fichier_etudiants IS NEW sequential_io(etudiant);
.......
```

Le paquetage *text_io* n'est pas générique. Il suffit donc d'une clause WITH pour y avoir accès et pouvoir faire des entrées sorties sur les caractères ainsi que sur les chaînes de caractères.

Remarquons que *text_io* contient des paquetages (sous-unités) génériques qui sont *integer_io*, *float_io* et *enumeration_io*.

Le paquetage *sequential_io* est générique. On y accède par WITH puis l'on instancie un exemplaire de ce paquetage.

## Primitives de gestion de fichiers

=> communes à *sequential_io*, *direct_io* et *text_io*.

=> Concernent la création, l'ouverture, la fermeture et la destruction.

```
Type file_type IS LIMITED PRIVATE;

Type file_mode IS (in_file, inout_file, out_file);
    -- le mode inout_file ne s'applique qu'aux fichiers à
    -- accès direct

❑ Procedure create(    file : IN OUT file_type;
                       mode : IN file_mode := out_file;
                       name : IN string := "";
                       form : IN string := "");
        -- form(format) dépend de l'implantation
```

- **Procedure open( file : IN OUT file_type;**
  **mode : IN file_mode := out_file;**
  **name : IN string := "";**
  **form : IN string := "");**

- **Procedure close (file : IN OUT file_type);**

- **Procedure delete (file : IN OUT file_type);**

- **Procedure reset (file : IN OUT file_type; mode : IN file_mode);**

- **Procedure reset (file : IN OUT file_type);**

- **Function mode(file : IN file_type) return file_mode;**

- **Function name(file : IN file_type) return string;**

- **Function form(file : IN file_type) return string;**

- **Function is_open(file : IN file_type) return boolean;**

## Désignation d'un fichier _

En ADA, les fichiers sont des objets d'un type *file_type* exporté du paquetage *text_io* ou d'un exemplaire de *sequential_io* ou *direct_io*.

Exemples :

• Pour faire des entrées/sorties séquentielles sur un fichier binaire contenant des informations sur les étudiants :

```
Type etudiant is record
      .....
End record;
Package fichier_etudiants IS NEW sequential_io(etudiant);
f_etudiant : fichier_etudiants.file_type;
```

• Pour faire des entrées/sorties séquentielles sur un fichier texte:

```
source : text_io.file_type;
```

*f_etudiant* et *source* seront ensuite le paramètre des opérations d'entrées sorties.

p *file_type* est un type limité privé. On ne peut donc pas faire d'opération d'affectation, tests d'égalité/inégalité sur des objets de ce type.

On ne peut que déclarer des objets de ce type et les passer en paramètre aux routines d'entrées/sorties. Un objet de ce type peut cependant être passé en paramètre en mode IN OUT (voir plus loin).

Comme dans la plupart des langages de programmation, un fichier ADA possède :

• un nom interne (logique) qui est du type *file_type* ci-dessus; c'est le paramètre FILE des routines d'entrées/sorties.

• un nom externe (physique). C'est le nom du fichier sur disque. Ce nom donné sous forme d'une chaîne de caractères est utilisé en paramètre des routines de création et d'ouverture.

## Utilisation

Afin d'utiliser un fichier, on établit un lien entre les noms logique et physique d'un fichier lors des opération CREAT et OPEN. Lors que ce lien est établi sans erreur, le fichier est dit ouvert et on peut le manipuler:

```
CREATE(equipe_de_foot, name => "/users/cnam/foot");
```

Avec cette opération, un fichier séquentiel dont le nom sut disque est */users/cnam/foot* est créé et sera manipulable dans le programme via le nom *equipe_de_foot*.

```
OPEN(source, IN_FILE, "test.a");
```

Ouverture d'un fichier texte qui a priori existe déjà sur le disque.

Le lien physique / logique est détruit lors de la fermeture (CLOSE) ou le destruction (DELETE) du fichier. Le fichier est alors dit fermé.

```
CLOSE(equipe_de_foot);
delete(source);
```

La fonction *is_open* permet de savoir si un fichier est ouvert ou fermé.

Si le nom (externe) d'un fichier est la chaîne nulle (“” : par défaut), le fichier est dit temporaire. La durée de vie d'un tel fichier est inférieure ou égale à celle du programme. Ces fichiers ne laisseront aucune trace sur le disque.

## Modes d'utilisation des fichiers

==> 3 modes :

- IN_FILE : lecture seule
- OUT_FILE : écriture seule
- INOUT_FILE : lecture et écriture. Ce mode n'est permis que pour les fichiers à accès direct.

A la création on ne peut préciser que les modes OUT_FILE ou INOUT_FILE.

La procédure RESET permet de modifier le mode d'ouverture d'un fichier déjà ouvert.

La fonction MODE permet de connaître le mode d'ouverture d'un fichier.

## Erreurs survenues lors d'utilisation des fichiers

Les trois paquetages prédéfinis text_io, sequential_io et direct_io fournissent les mêmes exceptions :

STATUS_ERROR : déclenchée quand un fichier n'est pas dans l'état voulu (déjà ouvert, fermé,..)

MODE_ERROR : déclenchée quand un fichier n'est pas dans le mode volulu (lecture sur un fichier ouvert en écriture seule...)

NAME_ERROR : déclenchée lors des opérations der création et d'ouverture lors que le paramètre NAME (nom externe) n'est pas correct, ou ouverture d'un fichier inexistant sur disque;

USE_ERROR : problème tels que la taille maxi atteinte, problèmes de protection...

DEVICE_ERROR : problèmes matériel du système d'entrées/sorties

END_ERROR : tentative de lecture à la fin du fichier

DATA_ERROR : problème du type de la valeur lue (lettre dans un entier..)

# A propos de END_ERROR

=> deux façons de lire un fichier :

Dans ce premier exemple, à la fin du fichier, on ne peut avoir qu'une fin de ligne ou une fin de page.

```
With text_io; use text_io;
With entier_io; use entier_io; -- paquetage instancié
Procedure exemple_1 IS
      i : integer; somme : integer := 0;
Begin
      While not end_of_file Loop
            get(i);
            somme := somme + i;
      End loop;
      put(somme); new_line;
End exemple_1;
```

Dans ce deuxième exemple, à la fin du fichier, des espaces peuvent apparaître après le dernier entier.

```
With text_io; use text_io;
With entier_io; use entier_io;
Procedure exemple_2 IS
      i : integer; somme : integer := 0;
Begin
      Loop
            get(i); somme := somme + i;
      End loop;
      -- Cette ligne n'est jamais atteinte
Exception
            When end_error =>  put(somme); new_line;
End exemple_2;
```

## A propos de NAME_ERROR

Ecrire une procédure qui tente d'ouvrir un fichier en écriture et s'il n'y parvient pas, tente de le créer.

```
Procedure ouvre_creation (nom_ada : IN OUT file_type;
                  nom_externe : string; acces : OUT boolean) IS
Begin
      open(nom_ada, out_file, nom_externe);
      acces := true;

Exception
      When name_error =>
            -- le fichier "nom_externe n'existe pas ou parce que le
            -- nom  fourni est incorrect. Dans ce cas, create
            -- déclenche la même exception
            Begin
                  -- l'usage d'un bloc permet de traiter l'exception
                  -- déclenchée par create. Sans ce bloc, l'exception
                  -- est propagée vers l'appelant.

                  create(nom_ada, name => nom_externe);
                  acces := true;
            Exception
                  When OTHERS => succes := false;
            End ;

      When OTHERS =>
                  -- on traite ici les cas tels que :
                  -- • fichier déjà ouvert (status_error)
                  -- • problème matériel (device_error)
                  -- • problème de protection (use_error)
            succes := false;
End ouvre_creation;
```

# Les fichiers Séquentiels

Le paquetage générique *sequential_io* fournit les primitives d'accès séquentiel à un fichier.

=> un fichier séquentiel ne peut être ouvert en lecture et écriture en même temps. (mode *inout_file* non autorisé).

En lecture, les élément sont accédés à partir du premier jusqu'au dernier. La fonction END_OF_FILE rend *true* lorsque la fin du fichier a été atteinte.

L'ouverture d'un fichier existant en écriture, ou l'opération RESET avec le mode *out_file* ont pour effet d'effacer le fichier; il est alors considéré comme vide.

- **Procedure read(file : IN file_type; item : OUT element_type);**

  **-- lecture de l'élément item**

- **Procedure write(file : IN file_type; item : IN element_type);**

  **-- écriture de l'élément item**

  **-- exceptions possibles:**

  **-- mode_error;**

  **-- use_error (si la taille maxi du fichier atteinte)**

- **Function end_of_file(file : IN file_type) return boolean;**

  **-- vrai si la fin du fichier atteinte en lecture.**

  **-- exception : mode_error (lecture seulement)**

# Compléments sur les fichiers textes

On a vu la manipulation des fichiers clavier et écran et les entrées/ sortie de chaînes, de caractères, d'entiers et de flottants.

Le paquetage *text_io* fournit quatre paquetages génériques :

- integer_io : pour les entrées/sorties d'entiers
- float_io : pour les entrées/sorties de flottants
- fixed_io : pour les entrées/sorties de réels fixes
- enumeration_io : pour les entrées/sorties de valeurs d'un type énuméré.

Pour faire ces entrées/sorties, il est nécessaire d'instancier ces paquetages.

Exemples d'instanciations :

```
Type couleur IS (bleau, vert, blanc, rouge...);
Type operateur IS ('+', '-', '/', '*', mod, abs);
Type abcisse IS range 0..80;
Type volt IS DELTA 0.125 Range 0.0 .. 255.0;

package coul_es IS new text_io.enumeration_io(couleur);
package oper_es IS new text_io.enumeration_io(operateur);
package abcisse_es IS new text_io.integer_io(abcisse);
package volt_es IS new text_io.fixed_io(volt);
```

## Entrées-sorties des entiers

Pour lire ou écrire un nombre entier, il faut instancier le paquetage générique *integer_io* interne à *text_io*.

On peut lire et écrire un entier (et un réel) dans une base quelconque comprise entre 2 et 16. La valeur par défaut est 10.

```
SUBTYPE numbre_base IS integer range 2..16;
default_base : number_base := 10;
```

Lecture d'entiers : La procédure GET :

```
SUBTYPE field IS integer range 0..X; --dépend de l'implantation
❑ Procédure GET(item : out integer; Width : field := 0);
```

Lorsque le champ Width n'est pas précisé (= 0); la procédure GET évite les espaces, fins de ligne et de page, jusqu'à trouver un début d'entier (signe ou chiffre). Elle lit alors un entier.

Lorsque le champ Width est différent de 0, Width caractères sont analysés dans la ligne courante (mois s'il ne reste pas autant de caractères).

Ecriture d'entiers : La procédure PUT :

```
default_width : field := integer'width;
❑ Procédure PUT(  item : IN integer;
                  width : IN field := default_width ;
                  base : IN number_base := default_base);
```

Put écrit item dans la base indiquée. Le nombre est cadré à droite dans un champ de longueur (au mois égal à) Width caractères.

Exemple :

Pour se positionner en <x,y> à l'écran (en mode terminal ANSI), on peut écrire la séquence "Escape [ x ; y H" :

```
With text_io; Use text_io;
With integer_io; use integer_io;
Procedure gotoxy(x,y : natural) IS
Begin
    put(ascii.esc);
    put(ascii.l_bracket);
    put(x,1);
        -- le paramètre Width vaut 1. Ainsi, les nombre d'un
        -- chiffre sont affichés sur un caractère, les nombre de 2
        -- chiffres sur 2 caractères,....
        -- Si l'on précise pas 1, le nombre est affiché à droite
        -- d'un champ de longueur dépendant de l'implantation
    put(ascii.semicolon);
    put(y,1);
    put("H");
End gotoxy;
```

exemple 21-1/2

## Entrées-sorties des réels

Pour lire ou écrire un nombre réel, il faut instancier le paquetage générique *float_io* (*fixed_io*) interne à *text_io*.

- **Procedure get(item : OUT float; width : IN field :=0);**

Lorsque le champ *Width* est égal à 0, la fonction *get* saute les espaces, les fins de ligne et de page, jusqu'à trouver un signe ou un chiffre. Elle lit alors le littéral réel. Lors que le champ *Width* est différent de 0, Width caractères sont analysés dans la ligne courante (ou moins s'il ne reste pas width caractères).

D'une manière générale, un littéral réel imprimé est sous la forme :

FORE.AFT E EXP

default_fore : field := 2;

default_aft : field := float'digits-1;

default_exp : field := 3;

- **Procedure PUT ( item : IN float;**
  **fore : IN field := default_fore;**
  **aft : IN field := default_aft;**
  **exp : IN field := default_exp);**

FORE : la partie entière

AFT : la partie fractionnaire

EXP : la partie exposant (peut être absente)

- Si EXP=0, le format de sortie est FORE.AFT

**Exemple :put(235.03, 5, 3, 0); -- "bb235.030"**

- Si EXP≠0 (par défaut), le format de sortie est FORE.AFT E EXP

==> La partie FORE contient un seul chiffre

**Exemple : dans "-1.2345670E+02", EXP=3, AFT=6, FORE=2**

## Entrées/Sorties de types énumérés

Le paquetage générique *enumeration_io* fournit les opérations de lecture et d'écriture de valeurs de type énuméré.

Une différence par rapport aux paquetage d'entrées sorties numériques est qu'il n'y a pas de type universel énuméré (comme integer ou float). De ce fait, l'instanciation du paquetage générique *enumeration_io* doit se faire après la déclaration du type énuméré qui fera l'objet des entrées/sorties.

La procédure GET permet de lire une valeur de type énuméré. Elle évite les espaces, les fins de ligne et de page se trouvant devant la valeur.

- **Procedure GET(item : OUT enum);**

- **Procedure GET(file : IN file_type; item : OUT enum);**

La procédure PUT permettant d'imprimer une valeur de type énuméré utilise deux paramètres :

• Width : similaire à celui de PUT sur les entiers. Si Width est supérieur au nombre de caractères formant la valeur à imprimer, l'impression est complétée par autant d'espaces à droite que la différence.

• Set : par défaut, la valeur est imprimée en majuscule (set=lower_case pour imprimer en minuscule).

- **Procedure PUT( item : IN enum;**
  **width : IN field := 0;**
  **set : IN type_set := upper_case);**

- **Procedure PUT( file : IN file_type;**
  **item : IN enum;**
  **width : IN field := 0;**
  **set : IN type_set := upper_case);**

==> Le paquetage fournit également une procédure GET permettant de lire une valeur à partir du début d'une chaîne :

- **Procedure GET( from : IN string;**
  **item : OUT enum;**
  **last : OUT positive);**

**-- from(last) donne la position du dernier caractère lu**
**-- de la chaîne from .**

Une procédure PUT permet d'écrire dans une chaîne :

- **Procedure GET( to : OUT string;**
  **item : IN enum;**
  **last : IN type_set := upper_case);**

```
-- cahier des charges :
-- Ecrire un programme qui lit deux entiers et un operateur;
-- fait l'operation et range les donnees lues + le resultat dans
-- un fichier de type texte

With text_io;

Procedure ENUM is

Type operateur is ('+','-','/','*',modu);

Package enum_io IS NEW text_io.enumeration_io(operateur);
Package int_io IS NEW text_io.integer_io(integer);

fic : text_io.file_type;
n1,n2 : integer;
op : operateur;
erreur : exception;

-- lecture de deux entiers et un opérateur
-- ATTENTION : les quatre premiers opérateurs sont données sous la
-- forme '+', '-, '*' et '/'  (avec les '' ) alors que modu est donne tel que

Procedure lire(N1,N2 : out integer; op : out operateur) is
Begin
      text_io.put("donner un entier, un operateur et un autre entier ");
      int_io.get(N1);
      enum_io.get(op);
      int_io.get(N2);
      text_io.skip_line;
EXCEPTION
   when text_io.data_error => text_io.put_line("problemes de donnes");
                              RAISE erreur;
   when others => text_io.put_line("problemes autres");
End lire;
```

```
function eval(N1,N2 : integer; op : operateur) return integer is
begin
      case op is
            when '+' => return (N1+N2);
            when '-' => return (N1-N2);
            when '*' => return (N1*N2);
            when '/' =>       if N2 /= 0 then return (N1/N2);
                              else raise erreur;
                               end if;
            when modu =>      if N2 /= 0 then return (N1 mod N2);
                              else raise erreur;
                              end if;
      end case;
End eval;

BEGIN
      text_io.create(fic,name => "data1");
      lire(n1,n2,op);

 ------------ affichage des donnees lues et le resultata l'ecran -------------
      int_io.put(n1,1); enum_io.put(op); int_io.put(n2,1);
       text_io.put(" ===> "); int_io.put(eval(n1,n2,op),1);text_io.new_line;

 ----------- ecriture des donnees et le resultat dans le fichier -----------
      int_io.put(fic,n1,1);
      enum_io.put(fic,op);
      int_io.put(fic,n2,1);
      text_io.put(fic," ===> "); int_io.put(fic,eval(n1,n2,op),1);
      text_io.new_line(fic);
      text_io.close(fic);

 EXCEPTION
      when erreur =>    text_io.put_line("problemes de donnes");
                        text_io.close(fic);
      when others =>    text_io.put_line("problemes autres");
                        text_io.close(fic);
End enum;
```

# Fichiers standard d'entrées/sorties

Nous avons examiné les fichiers standard d'entrées sorties. Les opération PUT et GET ne mentionnent pas explicitement de fichier. Au lancement d'un programme ADA, les fichiers par défaut sont les deux fichiers standard clavier et écran.

```
new_line;
put(5);
If End_Of_File THEN ...
```

ADA permet de modifier les canaux d'entrées sorties standard utilisés par les procédures SET_INPUT et SET_OUTPUT.

Le jeu d'opérations ci-dessous permet de manipuler ces fichiers :

- **Procedure set_input(file : IN file_type);**
  **-- le fichier par défaut en entrée devient FILE**
  **-- exception : status_error, mode_error**

- **Procedure set_output(file : IN file_type);**
  **-- le fichier par défaut en sortie devient FILE**
  **-- exception : status_error, mode_error**

Pour connaître les fichiers standard et courant

- **Function standard_input return file_type;**
- **Function standard_output return file_type;**
- **Function current_input return file_type;**
- **Function current_output return file_type;**

p L'affectation entre les objets de type file_type (limité privé) est interdite. On ne peut donc pas écrire :

```
fichier_courant := standard_input;
```

Pour réaliser le même effet :

```
set_input(fichier);
      -- fichier devient le fichier d'entrée par défaut

set_input(standard_input );
      -- le fichier standard_input devient le fichier d'entrée
      -- par défaut comme lors de lancement du programme
```

## Compléments sur les notions de ligne et de page

On peut fixer, pour un fichier de sortie, les valeurs de taille de ligne et de page.

==> Quand un fichier est ouvert en écriture, il n'y a pas de borne à la taille de ligne ou de page.

A l'aide des procédures et fonctions suivantes, on peut imposer une limite au nombre de caractères d'une ligne, ou de nombre de lignes d'une page. Le changement de ligne ou de page se fera alors automatiquement lors d'une opération PUT lorsque la valeur sortie ne tient pas sur la ligne courante :

• Modification de la longueur maximale d'une ligne, et d'une page :

- **Procedure set_line_length(file : IN file_type; to : IN count);**
- **Procedure set_line_length(to : IN count);**
- **Procedure set_page_length(file : IN file_type; to : IN count);**
- **Procedure set_page_length(to : IN count);**

•Interrogation de la longueur de la ligne et de la page

- **Function line_length(file : IN file_type) return count;**
- **Function line_length return count;**

- **Function page_length(file : IN file_type) return count;**
- **Function page_length return count;**

Pour les fichiers non bornés, les fonctions *line_length* et *page_length* renvoient :

```
unbounded : CONSTANT count := 0; -- longueur d'une ligne
```

Exemples

```
set_line_length(fich_texte,80);   -- limité à 80
set_line_length(fich_texte,unbounded);-- fichier non borné
```

## Exemples de manipulation de fichiers séquentiels

```
-- lecture et écriture de 10 entiers dans le fichier
-- attention : le contenu du fichier sera illisible

With sequential_io;
Procedure fichier1 IS
i : integer;
package mon_fic IS new sequential_io(integer);
use mon_fic;
fic : mon_fic.file_type;
Begin
      create(fic, name => "data");
      for j in 1..10 loop
            write(fic,j);
      End loop;
      close(fic);
End fichier1;
```

```
-- écriture de plusieurs enregistrements dans un fichier séquentiel
-- puis lecture et affichage de ces enregistrements

With text_io, integer_io;
Use text_io; integer_io;
With sequential_io;
Procedure fic_personne IS
type personne IS record
        nom,prenom : string(1..10);
        note : integer ;
        End record;
Package mon_fic IS NEW sequential_io(personne);
Use mon_fic;
fic : mon_fic.file_type;
p: personne := (nom =>(Others =>' '),prenom =>(Others =>' '),note =>0);

-- lecture des enregistrements du fichier et affichage à l'écran
-- on remarque l'utilisation de l'exception pour terminer proprement la
-- lecture. Un schéma classique == > while not end_of_file loop ...
-- provoque des décalage entre l'affichages et les lectures

Procedure lecture IS
Begin
        while not end_of_file(fic) loop
                read(fic,p);
                put("le nom "); put(p.nom);
                put(" le prenom "); put(p.prenom);
                put(" la  note "); put (p.note); new_line;
        End loop;
        EXCEPTION     -- plusieurs paquetages importent des
                                -- exceptions, on précise laquelle (en préfixant)
                                -- pour  lever l'ambiguïté
                When mon_fic.END_ERROR => put_line("fini");
                When OTHERS => put_line("problèmes de lecture");
End lecture;
```

```
-- lecture des enregistrements au clavier et écriture dans le fichier

procedure saisie IS
Begin
      put_line("saisie d'une série de coordonnes, fin = ^D ");

      LOOP      -- boucle infinie avec sortie par exception
            put("entrer un nom "); get(p.nom,i); skip_line;
            put("entrer un prenom "); get_line(p.prenom); skip_line;
            put("entrer une note "); get(p.note); skip_line;
            write(fic,p);
      End loop;

      EXCEPTION
            When mon_fic.END_ERROR => put_line("fini");
            When OTHERS => put_line("problèmes d'écriture");
End saisie;


BEGIN
      create(fic, name => "data");
      saisie;
      close(fic);

      open(fic, name => "data", mode => in_file);
      lecture;
      close(fic);
End fic_personne;
```

# Fichiers à accès direct

Notion d'index et de taille :

Avec un fichier à accès direct, il est possible d'accéder à un élément en fournissant sa position. Cette position est un entier ≥0 .

Un fichier à accès direct possède un index qui donne la position de l'élément courant. Lorsque le fichier est ouvert et positionné (RESET), l'index vaut 1; que le fichier soit vide ou non. L'index est incrémenté de 1 après chaque opération de lecture ou d'écriture.

L'index peut être modifié par la procédure suivante :

- **Procedure set_index(file : IN fiel_type; to : IN positive_count);**

Ainsi que par les procédure READ et WRITE (voir plus loin).

La valeur de l'index peut être obtenue par :

- **Function index(file : IN file_type) return positive_count;**

La taille d'un fichier à accès direct est la position de son dernier élément. Elle vaut 0 si le fichier est vide. On peut placer l'index au delà du dernier élément (Attention aux trous). On peut connaître la taille d'un fichier par la fonction :

- **Function size(file : IN file_type) return count;**

La fonction *end_of_file* rend true si l'index est strictement supérieur à la taille du fichier :

- **Function end_of_file(file : IN file_type) return boolean;**

## Opérations séquentielles sur les fichiers à accès direct

On peut alterner des accès direct et séquentiels sur un fichier à accès direct :

- **Procedure read(file : IN file_type; item : OUT element_type);**
- **Procedure write(file : IN file_type; item : IN element_type);**

C'est l'index qui fournit la position de l'élément lu ou écrit. Il est incrémenté après l'opération.

Il est possible de fournir aux procédures READ et WRITE un paramètre indiquant la position de l'élément à lire ou à écrire :

- **Procedure read(file : IN file_type; item : OUT element_type;**
  **from : IN positive_count);**

**-- *from* fournit la position de l'élément. Index=from+1 après**
**-- Exceptions : Mode_error, End_error, Data_error**

- **Procedure write(file : IN file_type; item : IN element_type;**
  **to : IN positive_count);**

**-- *to* fournit la position de l'élément. Index=to+1 après l'opération**
**-- Exceptions : Mode_error, use_error (capacité atteinte)**

p Seuls les fichiers à accès direct peuvent être ouverts à la fois en lecture et en écriture.

**Exemple** : le programme ci-dessous effectue la lecture d'un fichier à accès direct à partir d'une certaine position. Le fichier contient des informations sur les étudiant. Chaque information est imprimé à l'écran.

<< ex page 33-34>