

Analyse et programmation langage ADA



Informatique 2^{ème} année

R. Chelouah : rachid.chelouah@insa-toulouse.fr

Sommaire

- Chapitre I : Présentation
- Chapitre II : Unités lexicales
- Chapitre III : Types et sous types
- Chapitre IV : Ordres (Sélection, cas, itération, ...)
- Chapitre V : Sous programmes
- Chapitre VI : Tableaux (array)
- Chapitre VII : Chaînes de caractères (String)
- Chapitre VIII : Articles (record)
- Chapitre IX : Pointeur
- Chapitre X : Fichiers (File)
- Chapitre XI : Paquetages simples (package)
- Chapitre XII : Généricité
- Chapitre XIII : Tâches

- **Chapitre I** : **Présentation**
- **Chapitre II** : Unités lexicales
- **Chapitre III** : Types et sous types
- **Chapitre IV** : Ordres (Sélection, cas, itération, ...)
- **Chapitre V** : Sous programmes
- **Chapitre VI** : Tableaux (array)
- **Chapitre VII** : Chaînes de caractères (String)
- **Chapitre VIII** : Articles (record)
- **Chapitre IX** : Pointeur
- **Chapitre X** : Fichiers (File)
- **Chapitre XI** : Paquetages simples (package)
- **Chapitre XII** : Généricité
- **Chapitre XIII** : Tâches

Chapitre I : Présentation

- I.1 Nombres et leurs représentations en mémoire
- I.2 Analyse et programmation
- I.3 Spécificités du langage ADA
- I.4 Apprentissage d'un langage
 - Approche descendante
 - Approche ascendante
- I.5 Mise en page d'un programme

I.1 Nombres et leurs représentations en mémoire

- Les nombres-littéraux- ont une valeur déterminée par les signes qui les représentent.

- Un nombre littéral est soit :
 - Décimal (représenté en base 10)
 - Non décimal (bases 2 à 16 incluses)
 - Entier (sans partie fractionnaire)
 - Réel (avec partie fractionnaire, éventuellement nulle)

I.1 Nombres et leurs représentations en mémoire

Nombres entiers

Tout entier positif n peut s'écrire sous la forme :

$$n = c_q 2^q + c_{q-1} 2^{q-1} + \dots + c_1 2^1 + c_0 2^0$$

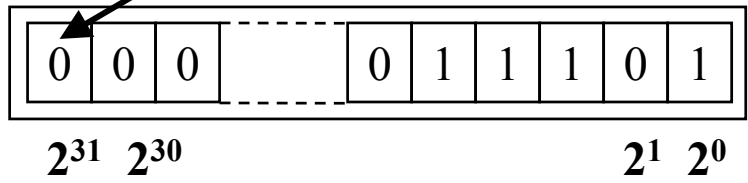
Si $c_q = 1$ ou 0 alors cette représentation est une représentation binaire du nombre n

Exemple le nombre 29 s'écrit en binaire **11101**

Écriture des entiers en binaire en mémoire

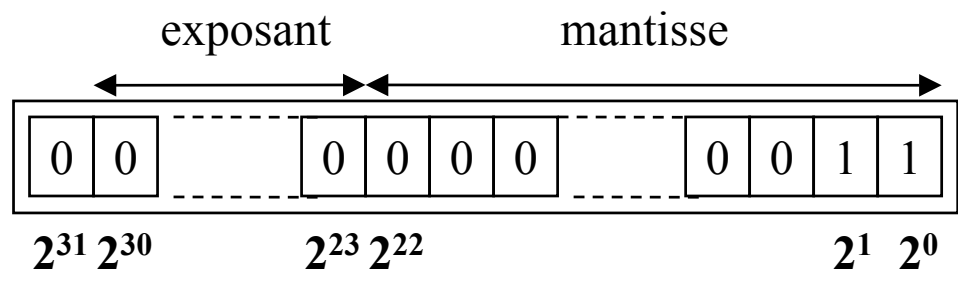
Le nombre 29 en binaire est stocké en mémoire de 32 bits

bit de signe (1 pour - et 0 pour +)



Nombres réels

Un nombre réel est représenté en machine sous forme d'une mantisse, d'un exposant et d'un bit de signe



I.2. Analyse et programmation

■ Algorithme et programmation

- Informatique (computer science)
- Algorithme (algorithm) est une suite d'opérations à effectuer pour résoudre un problème

■ Algorithme de résolution de l'équation $ax + b = 0$

- Si a est nul, l'équation est insoluble et l'algorithme est terminé
- Si a est non nul, transposé b dans l'autre membre, et l'équation devient $ax = -b$
- Diviser chaque membre par a et l'on obtient le résultat cherché est $x = -\frac{b}{a}$

■ *Retrait d'argent au distributeur*

■ *Appel téléphonique sur un mobile*

■ *Démarrage d'une voiture*

I.2. Analyse et programmation

- Algorithme de mise en marche d'une voiture
 - mettre la clé de le démarreur
 - serrer le frein à main
 - mettre le levier de vitesse au point mort
 - répéter les opération suivantes tant que le moteur ne tourne pas
 - ✓ mettre la clé dans la position *marche*
 - ✓ tourner la clé dans le sens des aiguilles d'une montre
 - ✓ attendre quelques secondes
 - ✓ si le moteur ne tourne pas, remettre la clé dans la position initiale
 - enclocher la première vitesse
 - desserrer le frein à main

- Une fois cet algorithme codé dans le langage de programmation, le programme ainsi créé doit être :
 - soit traduit complètement en langage machine par le **compilateur**
 - soit directement **interprété** (interpreted)

■ Méthode de décomposition par raffinements successifs

- Cette méthode est basée sur l'idée que, étant donné un problème à résoudre, il faut le décomposer en sous-problèmes de telle manière que
 - ✓ Chaque sous-problème constitue une partie du problème donné
 - ✓ Chaque sous-problème soit plus simple (à résoudre) que le problème donné
 - ✓ La réunion de tous les sous-problèmes soit équivalente au problème donné

➤ Exemple :

Construire une voiture → Construire la carrosserie

Construire le châssis

Construire le moteur → Construire la partie électrique

Construire la partie mécanique

I.3. Spécificités du langage ADA

■ Les avantages :

- Très proche de l'algorithmique
- Fortement typé
- Nombreuses vérifications faites par le compilateur
- Programmation modulaire obligatoire
- Structuration
- Abstraction (encapsulation, compilation séparée)
- Temps réel
- Interfaçage

 **une programmation plus propre avec moins d'erreurs**

■ Les inconvénients :

- Contraignant

I.3. Spécificités du langage ADA

- Deux types de fichiers en ADA :
 - Fichiers .ADB : ADA Body
Contiennent les corps du programme (équivalent au .c du C)
 - Fichiers .ADS : ADA Spécification
Contiennent les spécifications (équivalent au .h du C++)

- ADA manipule des objets. Qu'est qu'un Objet ?
 - Un objet est une constante ou une variable.
 - Un objet est typé.

- Qu'est qu'un Type ? ...
 - un ensemble de valeurs
 - un ensemble d'opérations « primitives » (sous-programmes) sur ces valeurs.

- ADA ne fait pas de différence entre minuscule et majuscule sur les noms des identificateurs

- Les commentaires en ADA :
 - Ceci est un commentaire qui
 - s'étend sur plusieurs lignes

I.3. Spécificités du langage ADA

■ Un programme ADA

Il comporte :

- un programme principal
- d'autres unités de programme
 - ✓ sous-programmes
 - ✓ paquetages

Le programme principal

- c'est une procédure
- appelle les services d'unité(s) de programme

I.4. Apprentissage d'un langage

- **Approche descendante** : On commence par un exemple et on essaie de l'analyser, et de comprendre la syntaxe et la sémantique
 - Va du composé au simple
 - Faite d'exemples simples
 - Vise à donner une connaissance globale (mais floue)
 - Méthode applicable aux langages faciles comme (Basic)

- **Approche ascendante** : décrire dans un ordre rigoureux la syntaxe, du plus élémentaire au plus général
 - Procède exactement en sens inverse
 - Vise à fournir des modes d'emploi précis
 - Méthode utilisable pour les langages simples dont la syntaxe est construite logiquement et sans trop de contraintes (Pascal)

- **Approche mixte** : Il est possible de recourir *alternativement* aux deux méthodes dans le cas où le langage est plus riche que rigoureux (langage C) ou *successivement* dans le cas où le langage est plus rigoureux que riche (ADA)

I.4. Apprentissage d'un langage

- 1.2.1 Éléments à traduire
- 1.2.2 Unités de programme
- 1.2.3 Types
- 1.2.4 Exemple
- 1.2.5 Dialogue programme utilisateur

I.4.1 Éléments à traduire

- l'algorithme principal
- les objets définis dans les lexiques
- les instructions élémentaires (affectation)
- les formes itératives
- les formes conditionnelles
- les formes d'analyse par cas
- les actions (procédures) et fonctions

I.4.2 Unités de programme

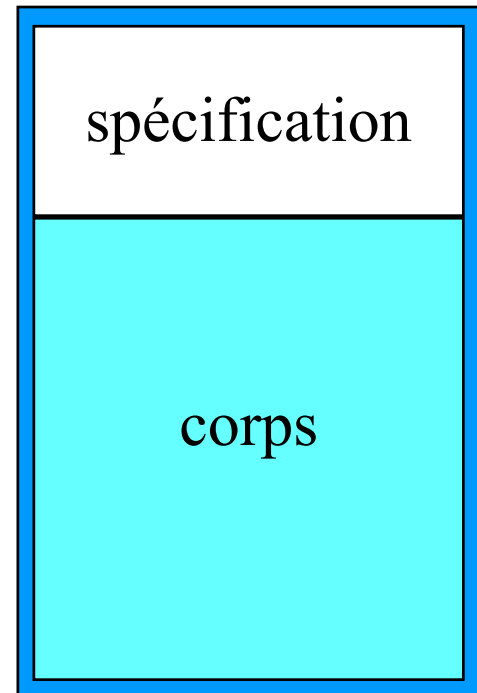
- Une unité de programme comporte deux parties :

- la partie déclaration

- ✓ spécification
 - ✓ élaborer les déclarations

- la partie instructions

- ✓ corps
 - ✓ exécuter les instructions



I.4.2 Unités de programme

■ Unité de programme

Notation algorithmique

lexique principal

définition des variables de
l'algorithme principal
et notification des
actions et des fonctions
utilisées

algorithme principal

texte de l'algorithme

ADA

- identification auteur, date
- définition de l'application
- utilisation des packages

-- *algorithme principal*

procedure AlgoPrinc is

-- lexique de l'algorithme principal

begin -- *AlgoPrinc*

-- *traduction de l'algorithme principal*

end AlgoPrinc ;

I.4.2 Unités de programme

with

inclusion de spécifications externes (paquetages)

**procedure, function, declare,
package body, task body**

spécification

begin

corps

end;

I.4.2 Unités de programme

```
with Text_IO ; -- Appel aux bibliothèques
```

```
procedure Afficher is
```

```
-- Partie Déclaration
```

```
begin --Afficher
```

```
-- Partie Instructions
```

```
exception
```

```
when ... => -- Traite-exception
```

```
end Afficher;
```

-
- *Fichier : Afficher.adb*
 - *Application : Hello world!*
 - *Auteur : Alexandre Meyer*
 - *Liste des packages utilisés par notre programme*
-

with Text_IO;

procedure Afficher **is**

begin -- *Afficher*

Text_IO.Put(" Salut tout le monde !");

exception -- *traitement des exceptions*

when others => Text_IO.Put_Line ("Erreur dans le programme principal");

end Afficher;

I.4.3 Types

- Un **type** définit les valeurs que peut prendre un objet et les opérations qui lui sont applicables. Il existe 4 grandes classes : les types *scalaires* ; les types *composés*; les types *privés*; les types d'*accès*; les types *dérivés*

Les types scalaires, peuvent être soit *discrets* (entier, énumérés), soit *numériques réels*.

Notation algorithmique

ADA

N : entier { définition }

B : booléen { déf. }

C : caractère { déf. }

R : réel { définition }

CH : chaîne { déf. }

N : Integer; -- *définition*

B : Boolean; -- *définition*

C : Charater; -- *définition*

R : Float; -- *définition*

CH : Unbounded_String; -- *définition*

Remarque: Il existe d'autres types pour les chaînes.

■ Exemple

```
-- Fichier : hello.adb
-- Application : Hello world!
-- Auteur : Alexandre Meyer
-- Liste des packages utilisés par notre programme
```

```
with Ada.Text_IO;  -- package d'entrée/sortie de texte (clause de contexte)
use Ada.Text_IO;
```

```
-- algorithme principal
```

```
procedure Afficher_Hello is
```

commentaire
de bloc

```
  -- Partie déclarative
```

```
  I : Integer := 5;      -- une variable qui ne sert à rien!!
```

```
begin -- Afficher_Hello
```

```
  -- Partie Instruction, Bloc, Corps
```

```
  Put_Line("Hello world!");
```

```
  Put(" Bye" );
```

commentaire
de ligne

```
end Afficher_Hello;
```

- **Dialogue programme-utilisateur => interface homme-machine**

- Présentation
- Lecture de nombres entiers ou réels
- Passage à la ligne lors de la lecture
- Mise en page du texte affiché par un programme
- Passage à la ligne lors de l’affichage
- Abréviations d’écriture

■ **Présentation**

➤ pour l'utilisateur:

- ✓ l'introduction des données;
- ✓ la commande du logiciel;
- ✓ la compréhension du déroulement des opérations;
- ✓ la compréhension des résultats obtenus;

➤ pour le programme:

- ✓ la demande des données nécessaires à son exécution;
- ✓ la production de résultats lisibles et clairement présentés;
- ✓ la quittance des opérations importantes effectuées;
- ✓ la mise en garde de l'utilisateur en cas de donnée erronée;
- ✓ la mise en garde de l'utilisateur en cas de commande erronée ou dangereuse.

I.4.5 Dialogue programme-utilisateur

- Lecture de nombres entiers ou réels

```
-- afficher un message clair à l'utilisateur pour lui indiquer ce qu'il doit faire;
-- attendre que l'utilisateur ait introduit la valeur;
-- lire la valeur;
-- reprendre son exécution.
```

```
with Ada.Text_IO;           -- paquetage d'entrée/sortie pour texte
with Ada.Integer_Text_IO;  -- paquetage d'entrée/sortie pour les entiers
with Ada.Float_Text_IO;    -- paquetage d'entrée/sortie pour les réel
-- ...
-- Calcul de la moyenne
procedure Calculer_Moyenne is
    Poids_Maths   : Integer : 3 ;    -- Coefficient des mathematiques
    Poids_Info    : Integer : 2;     -- Coefficient de l'informatique
    Note_Maths    : Float;           -- Note des mathematiques
    Notes_Info    : Float;           -- Note de l'informatique
    Moyenne       : Float;           -- Moyenne de l'eleve
    ...
    -- Autres declarations...
```

I.4.5 Dialogue programme-utilisateur

begin -- *Calculer_Moyenne*

-- *Presentation du programme...*

-- *Obtenir les notes d'un etudiant*

Ada.Text_IO.Put ("Donnez la note des mathematiques : ");

Ada.Float_Text_IO.Get (Note_Maths);

Ada.Text_IO.Put ("Donnez la note d'informatique : ");

Ada.Float_Text_IO.Get (Note_Info);

Ada.Text_IO.New_Line; -- *New_Line(3) sauter 3 lignes*

-- *Calcul de la moyenne*

Moyenne := (Note_Maths * Float(Poids_Maths) + Note_Info*Float(Poids_Info)) /
 (Float(Poids_Maths + Poids_Info));

-- *Montrer a l'utilisateur la valeur de sa moyenne*

Ada.Text_IO.Put (" Votre moyenne est de : ");

Ada.Float_Text_IO.Put (Moyenne);

end Calculer_Moyenne ;

I.4.5 Dialogue programme-utilisateur

Programme	Ce que l'utilisateur a donné
Ada.Float_Text_IO.Get(Note_Maths) ; Ada.Float_Text_IO.Get(Note_Info) ;	4.5 3 4.1

■ Passage à la ligne lors de la lecture

Programme	Ce que l'utilisateur a donné
Ada.Float_Text_IO.Get(Note_Maths) ; Ada.Text_IO.Skip_Line ;	4.5 3 <i>-- action de vider le buffer d'entrée</i>
Ada.Float_Text_IO.Get(Note_Info) ;	4.1

- Mise en page du texte affiché par un programme

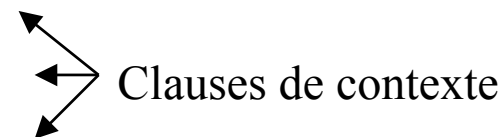
```
Ada.Integer_Text_IO.Put ( Poids_Maths+Poids_Info , 2 );
Ada.Float_Text_IO.Put ( Moyenne, 2, 1, 1 );
```

- Passage à la ligne lors de l'affichage

```
Ada.Text_IO.Put ( "Avec les notes que vous avez : " );
Ada.Float_Text_IO.Put ( Note_Maths );
Ada.Text_IO.Put ( " & " );
Ada.Float_Text_IO.Put ( Note_Info);
Ada.Text_IO.New_Line;
Ada.Text_IO.Put ( "Votre moyenne vaut : " );
Ada.Float_Text_IO.Put ( Moyenne , 2, 1, 1 );
Ada.Text_IO.New_Line;
```

■ Abréviations d'écriture

```
with Ada.Text_IO;           use Ada.Text_IO;
with Ada.Integer_Text_IO;  use Ada.Integer_Text_IO;
with Ada.Float_Text_IO;   use Ada.Float_Text_IO;
```



Clauses de contexte

-- Calcul de la moyenne

procedure Calculer_Moyenne **is**

```
  Poids_Maths   : Integer : 3 ;    -- Coefficient des mathematiques
  Poids_Info    : Integer : 2;    -- Coefficient de l'informatique
  Note_Maths    : Float;          -- Note des mathematiques
  Notes_Info    : Float;          -- Note de l'informatique
  Moyenne       : Float;          -- Moyenne de l'eleve
```

begin *-- Calculer_Moyenne*

```
  -- Presentation du programme...
  -- Obtenir le moyenne des notes d'un etudiant
  Put ( "Donnez la note des mathematiques : " );
  Get (Note_Maths);
  Put ( "Donnez la note d'informatique : " );
  Get (Note_Info);
```

I.5. Mise en page d'un programme

- Introduction
- En-tête des programmes et sous-programmes
- Entrées - Sorties
- Types
- Déclarations des variables
- Format des identificateurs et mots réservés
- Mise en forme du code
- Commentaires

I.5. Mise en page d'un programme

Il est indispensable de présenter une certaine rigueur dans la manière d'écrire un programme :

- Dans un cadre industriel, afin de simplifier la communication et accroître la productivité
- Dans un cadre scolaire, afin de prendre de bonnes habitudes et de simplifier le travail de correction

Le formalisme exact de la présentation des commentaires, dans ses détails, dépend évidemment du langage. Nous prendrons la forme ada où le symbole "--" débute un commentaire

■ Exemple non mis en forme

```

with Ada.Text_IO;           with Ada.Float_Text_IO;
use Ada.Text_IO;           use Ada.Float_Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

```

```

procedure Calcul is

```

```

  J : Integer ; G , P : float;

```

```

begin

```

```

  Get(J); Get(G);

```

```

  P := G / Float(J); Put(P);

```

```

end Calcul;

```


I.5. Mise en page d'un programme

- Utiliser un en tête pour chaque fichier utilisé

- Commenter chaque déclaration ou instruction importante en indiquant la **raison** de sa présence. Ce commentaire se place avant ou à coté de la déclaration ou de l'instruction;

- Choisir des identificateurs faciles à comprendre ;

- Assurer un degré élevé de systématique et de cohérence ;

- Effectuer une seule déclaration ou instruction par ligne ;

- Indenter, c'est-à-dire décaler vers la droite les déclarations ou les instructions contenues dans une autre déclaration ou instruction

■ En tête d'un programme

```

-- Nom fichier      : Calcul_Gain.adb
-- Auteur           : Dupond Jean
-- Date             : 26/10/2002
-- But              : Afficher le gain de chaque joueur  (description)
-- Dates de modif.  : uniquement s'il y a des modifications
-- Raison           :
-- Modules appeles  : Text_IO, Float_Text_IO, Integer_Text_IO
-- Mat. Particulier : matériel nécessaire pour l'utilisation de ce
--                  module.
-- Environnement    : JRASP
-- Compilation      : GNAT (ADA95)
-- Mode d'execution : Console

```

■ En tête d'un sous programme

```
-- Nom           : Calculer_Somme
-- But           : Additionner un nombre à un autre (Somme =Somme + Nombre)
--               : La somme ne doit pas dépasser une certaine limite
-- In            : Nombre a additionner
-- In            : Limite a ne pas dépasser
-- Out           : Somme de la somme précédente et du nombre
-- Out           : La somme dépasse-t-elle la limite ?
```

```
procedure Calculer_Somme ( Somme      : in out Integer;
                          Nombre     : in      Integer;
                          Limite     : in      Integer;
                          Depassement : out    Boolean) is
```

- Le nom d'une procédure devrait être toujours à l'infinitif
- Il est raisonnable de donner le même nom à un programme et au fichier qui le contient
- Il est déconseillé de mettre des caractères accentués dans les commentaires

I.5. Déclaration des variables

Dans tous les programmes, modules et sous-programmes, et quel que soit le langage, chaque variable :

- doit posséder un nom significatif
- est expliquée par un commentaire
- en général la seule présente sur une ligne

L'explication se met si possible à côté de la déclaration

Exemples :

Borne_Inf : Integer ; -- limite inférieure du traitement

Borne_Sup : Integer ; -- limite supérieure du domaine de définition

■ Format des identificateurs et mots réservés

Les mots réservés sont **toujours en minuscules**. Les identificateurs par contre doivent avoir la première lettre en majuscule. Si l'identificateur est composé de plusieurs mots, on les sépare avec un '_' et chaque mot commence par une majuscule.

Exemples :

Mots réservés : **procedure** **begin** **end** **loop**

Identificateurs : Borne_Inf Afficher Text_IO Enumeration_IO

Note : un identificateur est le nom de quelque chose comme (constante ou variable, type, exception, procédure ou fonction, attribut, paquetage, ...).

I.5. Mise en page d'un programme

■ Mise en forme du code

L'indentation correcte du code facilite énormément la lecture. Beaucoup d'erreurs proviennent d'une mauvaise indentation.

Il est déconseillé d'utiliser des tabulations pour faire l'indentation. Si vous changez d'éditeur de texte et que la taille de la tabulation est différente, toute votre mise en page est à refaire.

■ Alignement

L'alignement des instructions est aussi important pour la lisibilité du code.

Afin de bien distinguer les variables et les types, on aligne généralement les ":" et les ":=" des déclarations.

Exemple :

```

Resultat : Integer;
Nombre1  : Integer  := 0;
PI       : Float    := 3.1415;
    
```

I.5. Mise en page d'un programme

■ Commentaires

Les commentaires se mettent lors de l'écriture du code et non après

Un commentaire expliquant une partie de programme vient avant cette partie.

Un commentaire qui n'apporte aucune information ne sert à rien.

Exemple :

<pre>-- Incrementer I de 1 I := I + 1;</pre>	<pre>-- Passage à l'element suivant I := I + 1;</pre>
<pre>-- Si I plus grand que 0 alors if I > 0 then ...</pre>	<pre>-- Si l'indice existe if I > 0 then ...</pre>

-- Presentation d'un programme

```
Put_Line(" calcul de la surface d'un cercle ");
```

-- Boucle d'affichage de l'alphabet minuscule

```
for I in 'a'..'z' loop
```

```
    Put(I);
```

```
    New_Line;
```

```
end loop;
```

I.5. Mise en page d'un programme (Exemple bien présenté)

```

with Ada.Text_IO;          use Ada.Text_IO;          -- Module d'entree/sortie de texte
with Ada.Float_text_IO;    use Ada.Float_text_IO;    -- Module d'entree/sortie de reels

```

```

procedure Calculer_Benefice is

```

```

    Prix_Achat : Float;
    Prix_Vente : Float;
    Recette    : Float;

```

```

begin -- Calculer_Benefice

```

```

    Put(" Donner le prix d'achat : ");

```

```

    Get(Prix_Achat);

```

```

    Skip_Line;

```

```

    Put_Line(" Donner le prix de vente : ");

```

```

    Get(Prix_Vente);

```

```

    Skipe_Line;

```

```

    Recette:= Prix_Vente - Prix_Achat;

```

```

if (Recette >0.0) then Put(" Nous avons fait un benefice ");

```

```

else Put(" Nous avons des pertes ");

```

```

end if;

```

```

end Calculer_Benefice;

```


Sommaire

- Chapitre I : Présentation
- **Chapitre II : Unités lexicales**
- Chapitre III : Types et sous types
- Chapitre IV : Ordres (Sélection, cas, itération, ...)
- Chapitre V : Sous programmes
- Chapitre VI : Tableaux (array)
- Chapitre VII : Chaînes de caractères (String)
- Chapitre VIII : Articles (record)
- Chapitre IX : Pointeur
- Chapitre X : Fichiers (File)
- Chapitre XI : Paquetages simples (package)
- Chapitre XII : Généricité
- Chapitre XIII : Tâches

Chapitre II : Unités lexicales

- II.1 Introduction
- II.2 Exemple d'unités lexicales

II.1 Introduction

-- ADA ne fait pas de différence entre minuscule et majuscule

- **Jeu de caractères en Ada :**
 - **LATIN-1** qui contient l'ASCII
 - Caractères imprimables et non imprimables
- **Unités lexicales** (*lexical units*) Mots, Vocabulaire
- **Unités syntaxiques** (*syntactic units*) Phrases
- **mots et symbole de ponctuation**

Ada	est	un	langage
nom sujet	verbe verbe	article	nom complément

II.2 Exemple d'unités lexicales

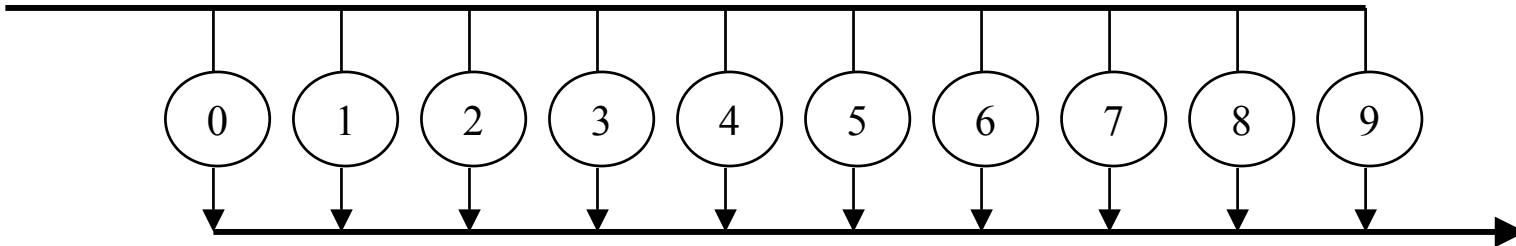
Une *unité lexicale* (mot d'un langage de programmation), est le plus petit élément d'un programme, non décomposable.

- **Identificateurs** : Haut, Jaune...
- **Mot réservés** : **begin, end...**
- **Identificateurs prédéfinis** : Integer, Float
- **Nombre entiers et nombre réel** : 12, 15.3
- **Commentaires** : -- ceci est un commentaire..
- **Constantes numériques** : 13, 13.6
- **Constantes caractères** : 'a', 'A', '1', '+'..
- **Constantes chaînes de caractères** : "blabla", "1", "", " "
- **Délimiteurs** : +, -, *, :=, >=

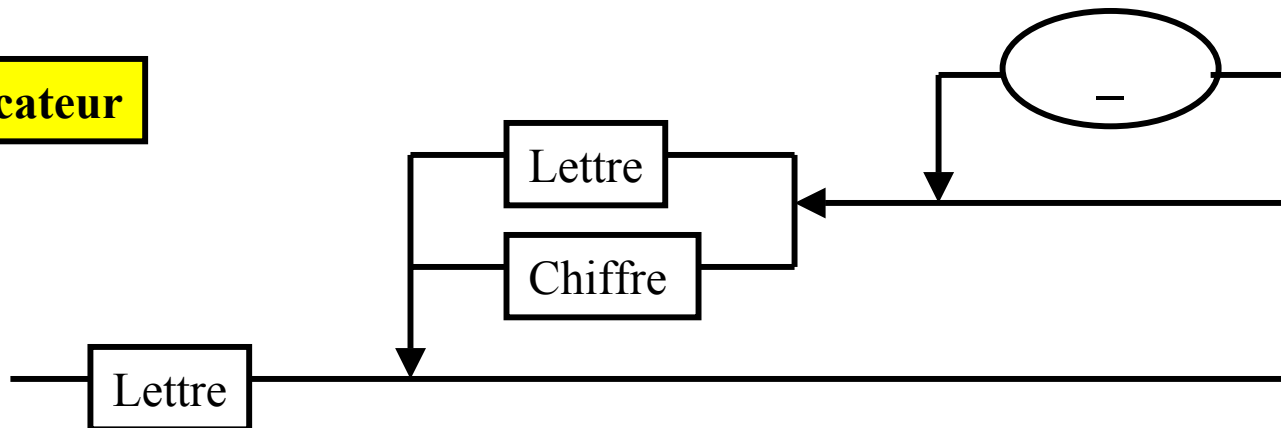
Identificateurs (*ou id.*), mot choisie

- le schéma lexical d'un identificateurs (*ou id.*) ou mot choisie est :

Chiffre



Identificateur



PI

Epsilon

Valeur_absolue

V1

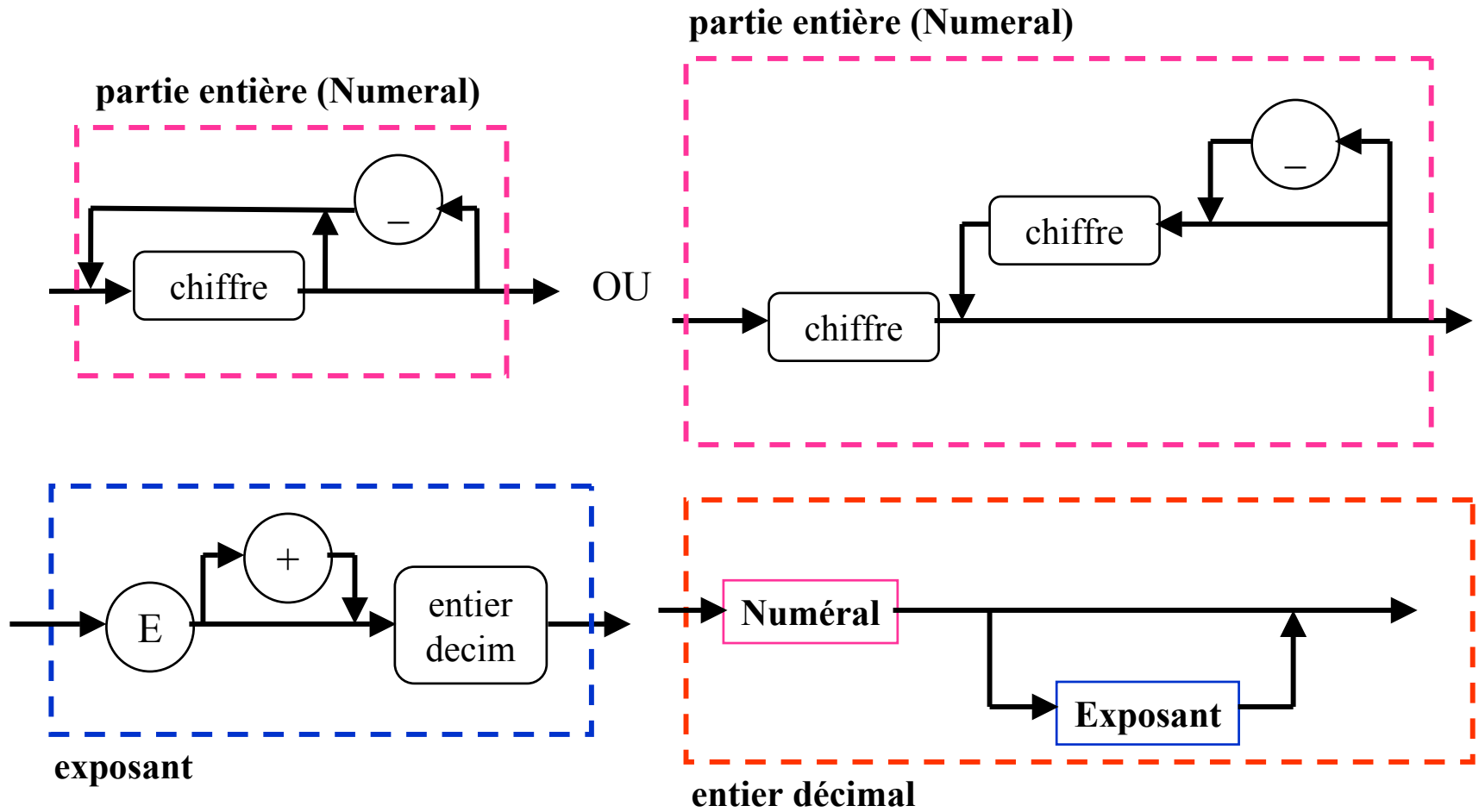
V_1

Sont des identificateurs différents

Identificateurs, mots réservés

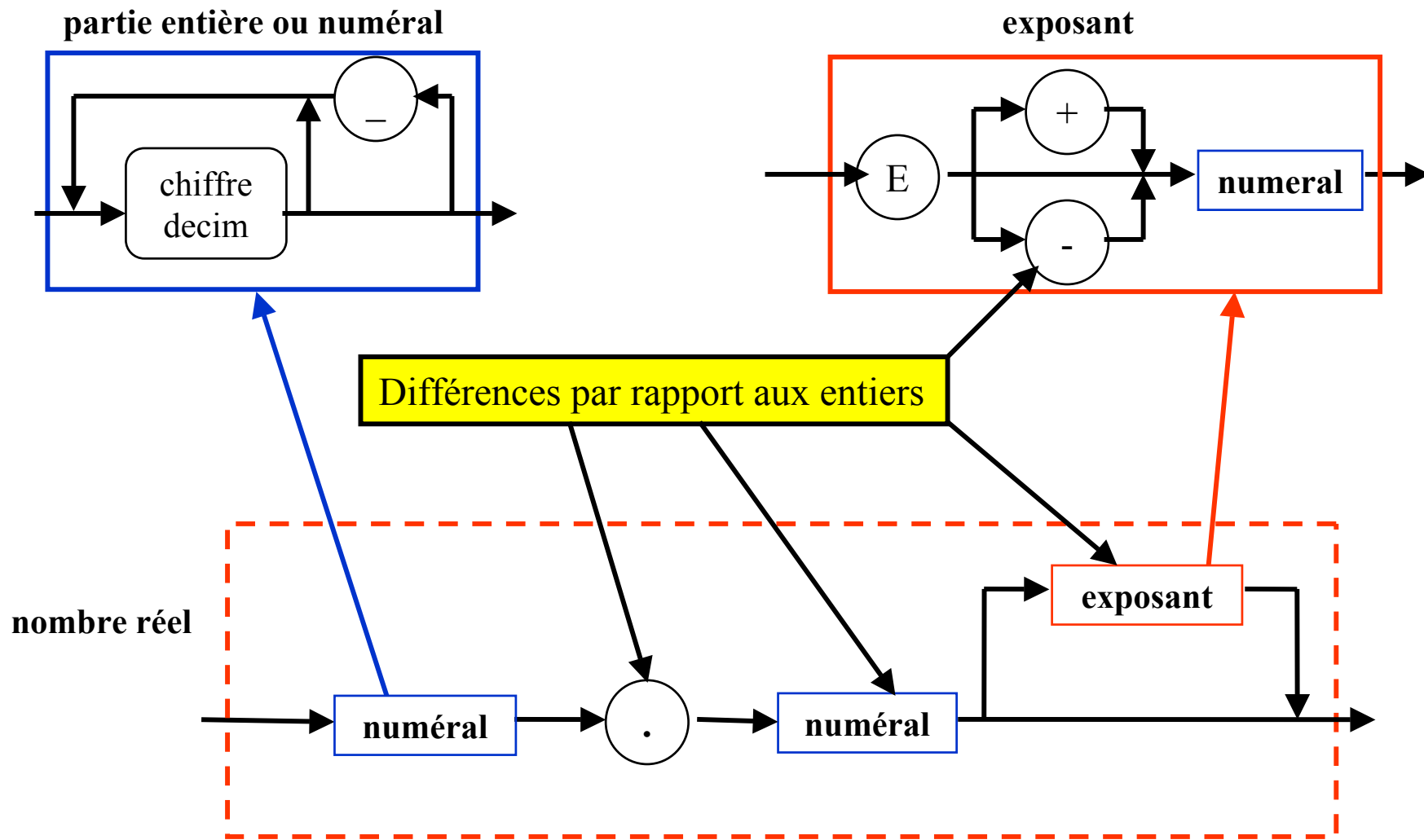
ABORT	ACCEPT	ACCESS	ALL	AND
ARRAY	AT	BEGIN	BODY	CASE
CONSTANT	DECLARE	DELAY	DELTA	DIGITS
DO	ELSE	END	ENTRY	EXCEPTION
EXIT	FOR	FUNCTION	GENERIC	GOTO
IF	IN	IS	LIMITED	LOOP
MOD	NEW	NOT	NULL	OF
OR	OTHERS	OUT	PACKAGE	PRAGMA
PRIVATE	PROCEDURE	RAISE	RANGE	RECORD
REM	RENAMES	RETURN	REVERSE	SELECT
SEPARATE	SUBTYPE	TASK	TERMINATE	THEN
TYPE	USE	WHEN	WHILE	WITH
XOR				

Nombre entier (décimal)



■ Exemple : 12 0 123_456
 1 E 2 1 E +2 (identiques à 100)

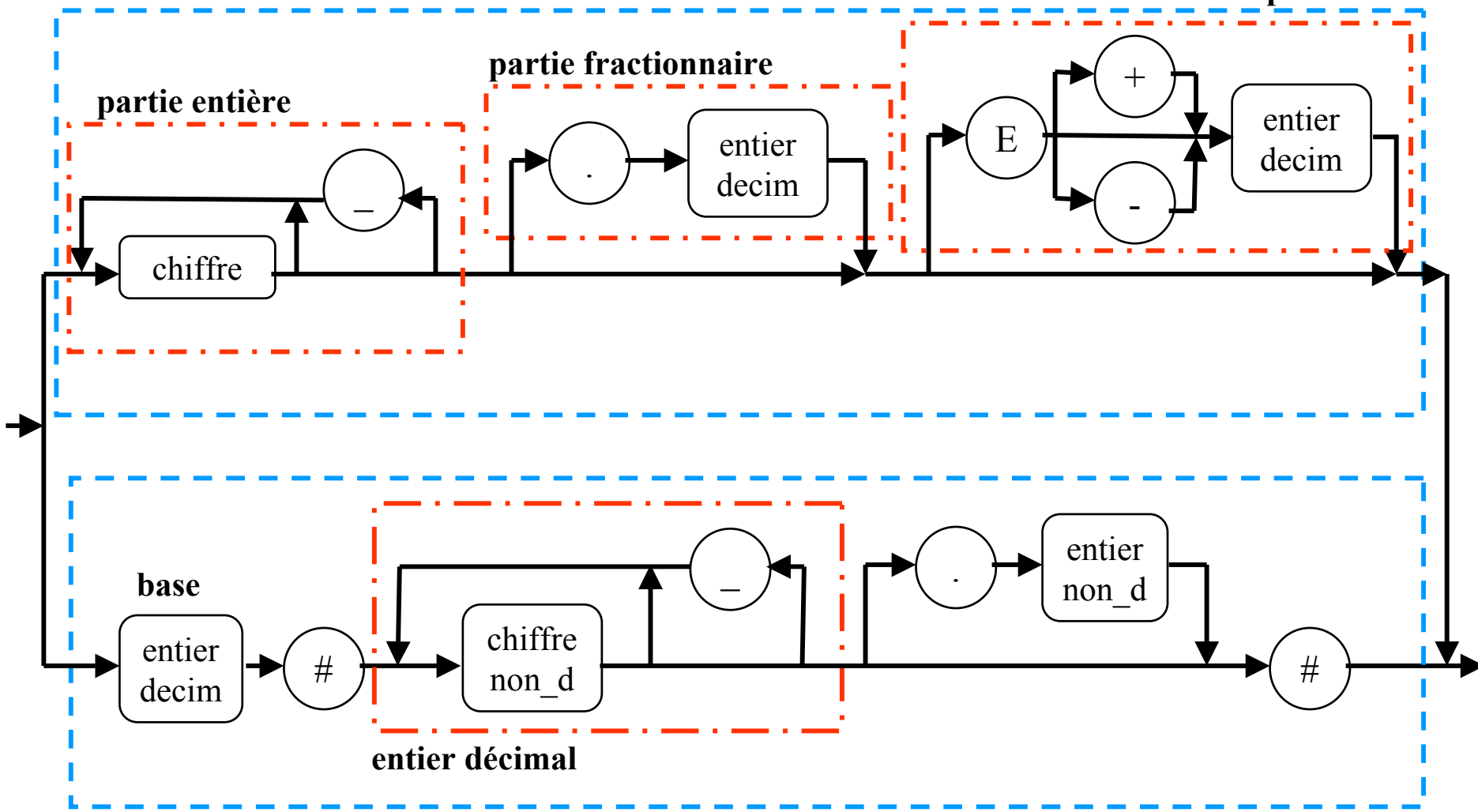
Nombre réel (décimal)



■ Exemple

12.0	0.0	3.14159_26535_89793
314.0 E -2	0.314E+1	(identiques à 3.14)
noter que	314E-2	est erroné (314 entier)

Nombre décimal (entier ou réel)



Nombre non décimal (entier ou réel)

Résumé des nombres-littéraux

- Entiers décimaux

12	0	123_456
1 E 2	1 E +2	(identiques à 100)

- Réels décimaux

12.0	0.0	3.14159_26535_89793
314.0 E -2	0.314E+1	(identiques à 3.14)
noter que	314E-2	est erroné

- Entiers non décimaux

2#1010#	16#A#	(valant 10)
2#0111_000#	16#70#	(valant 112)
2#111#E4	16#7#E1	(valant 112)

- Réels non décimaux

2#1.1#	(valant 1.5)
16#A.6#	(valant 10 et 3/8)
2#1.0#E2	(valant 4)
16#A.0#E1	(valant 160)

Déclaration de constantes

- `Un : constant Integer :=1; --Constante entière`
- `Dix_Mille : constant Integer := 10_000 ;`
- `PI_2 : constant Float := PI/2.0 ;`
- `Initial : constant Character := 'A' ;`
- `Str : constant STRING := "ADA" ;`
- `Point : constant Vecteur := (10,-20,100); -- un agrégat de type vecteur`

Délimiteurs

Deux unités lexicales sont isolées et reconnues comme distinctes si et seulement si un délimiteur est placé entre elles, c'est à dire si :

- Un (ou plusieurs) blanc(s) les sépare(nt) ; dans une chaîne de caractères le rôle du séparateur du blanc est annulé ;
- Ou bien un (ou plusieurs) retour(s) à la ligne est (sont) effectués ;
- Ou bien elles sont séparées par un caractère spécial, ou une combinaison de deux caractères spéciaux, figurant dans cette liste :

&	=	'	>
(!)	=>
+	..	-	**
*	:=	/	/=
.	>=	,	<=
:	<<	;	>>
<	<>		

Remarque : # , " , --, ne sont pas des délimiteurs, ils font partie d'unités lexicales.

Sommaire

- Chapitre I : Présentation
- Chapitre II : Unités lexicales
- **Chapitre III : Types et sous types**
- Chapitre IV : Ordres (Sélection, cas, itération, ...)
- Chapitre V : Sous programmes
- Chapitre VI : Tableaux (array)
- Chapitre VII : Chaînes de caractères (String)
- Chapitre VIII : Articles (record)
- Chapitre IX : Pointeur
- Chapitre X : Fichiers (File)
- Chapitre XI : Paquetages simples (package)
- Chapitre XII : Généricité
- Chapitre XIII : Tâches

Chapitre III : Types, sous-types et nouveaux types

- III.1. Introduction
- III.2. Classification des types de données en ADA
- III.3. Présentation d'un type discret
 - III.3.1. Présentation d'un type entier
 - III.3.2. Présentation d'un type énumération
 - III.3.2.1. Caractère
 - III.3.2.1. Booléen
- III.4. Présentation d'un type réel
- III.5. Utilisation des types
- III.6. Contraintes
 - III.6.1. La contrainte de domaine
 - III.6.2. La contrainte de précision
 - III.6.3. La contrainte d'indice
 - III.6.4. La contraintes de discriminant
- III.7. Sous type
- III.8. Nouveaux types
- III.9. Type dérivé

Un *type* caractérise un ensemble de valeurs et les opérations définies sur cet ensemble.

■ Objectifs du typage

- Fournir une structure et des propriétés aux données
- Vérifier leur intégrité dans tout le programme
- Éviter de mélanger accidentellement les données qui ne sont pas comparables
- Toute violation de type sur les objets est sanctionnée par le compilateur

■ Aucun objet ne peut recevoir de valeur ni subir une opération si l'ensemble auquel il appartient n'a pas été préalablement déterminé c'est à dire qu'il faut :

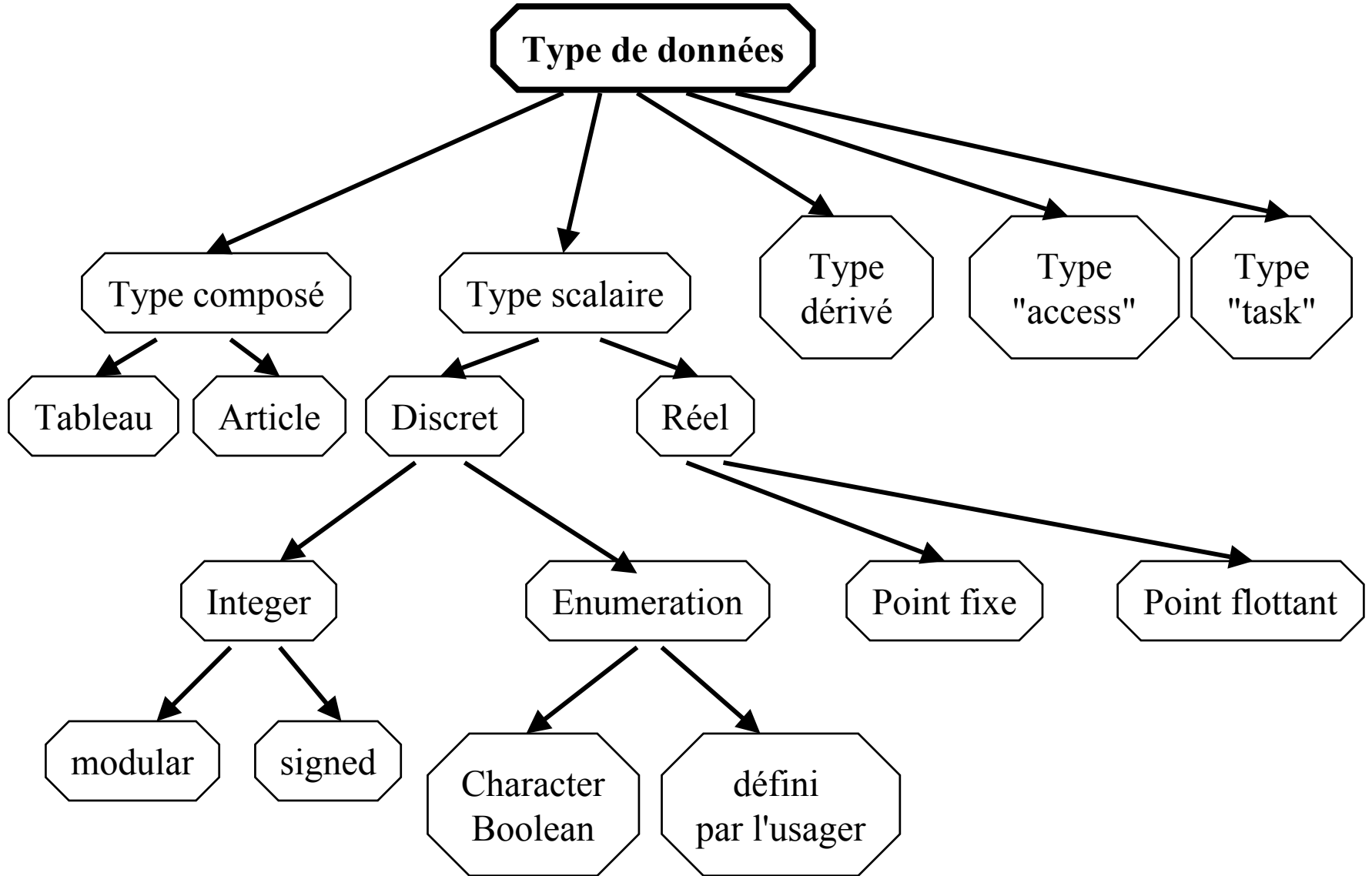
- présenter un type et le déclarer
- déclarer que l'objet est de ce type.

■ Un *sous-type* d'un type (connu) caractérise un *sous-ensemble* des valeurs associées à ce type, les opérations étant les mêmes.

Définir un sous type demande :

- que le type de base soit présenté et déclaré
- que le sous-type soit défini par une *contrainte* restreignant l'ensemble des valeurs.

III.2 Classification des types de données en ADA



III.3 Présentation d'un type discret

■ Pour commencer, un peu de vocabulaire: les "types discrets" sont les types ordonnés où chaque élément a un successeur et un prédécesseur. Ils sont munis d'attributs, des fonctions exprimant les propriétés courantes du type. La syntaxe est T 'nom_attribut, où T est un type discret et X une variable du type T . Ces attributs sont les suivants:

- T'Base Type de base de T
- T'FIRST Borne inférieure du type T .
- T'LAST Borne supérieure du type T .
- T'POS(X) X est de type T et T'POS(X) renvoie la position de X dans le type T .
- T'VAL(N) N est de type entier et T'VAL(N) est l'élément de position N dans le type T .
- T'SUCC(X) X est de type T et T'SUCC(X) est le successeur de X dans T , ainsi $T'SUCC(X) = T'VAL(T'POS(X)+1)$.
- T'PREC(X) X est de type T et T'PREC(X) est le prédécesseur de X dans T , ainsi $T'SUCC(X) = T'VAL(T'POS(X)-1)$.
- T'IMAGE(X) X est de type T et T'IMAGE(X) est une chaîne de caractères qui contient l'image de X .
- T'Min(X , Y) donne le minimum entre X et Y
- T'Max(X , Y) donne le maximum entre X et Y
- T'WIDTH Donne le nombre maximal de caractères pour une image d'une valeur de T .
- T'VALUE(S) S est une chaîne de caractères qui est l'image d'un élément de type T ; T'VALUE(S) est l'élément en question.

II.3.3.1 Type entier (opération)

- Le type discret entier INTEGER. Il possède deux sous-types prédéfinis: NATURAL pour les entiers naturels et POSITIVE pour les entiers strictement positifs. Certains opérateurs sont prédéfinis, ce sont l'addition +, la multiplication *, la soustraction -, la division /, le reste de la division mod, l'exponentiation **.

- **les opérateurs arithmétiques**
 - les opérateur unaires + - abs
 - les opérateurs binaires + - * / ** rem mod
 - ✓ Le mot réservé **rem** représente le reste de la division entière (euclidienne).
 - ✓ Notons les relations suivantes : $A \text{ rem } (-B) = A \text{ rem } B$ et $(-A) \text{ rem } B = -(A \text{ rem } B)$.
 - ✓ Le mot résevée **mod** représente l'opération mathématique *modulo* qui ne sera pas détaillée ici.
 - ✓ les expressions entières sont des combinaisons de ces opérations. Il faut alors prendre garde au fait qu'une telle expression n'est toujours pas calculable !

- **Priorité des opérateurs arithmétiques**
 - les opérateurs ** et **abs**;
 - les opérateurs binaires * / **rem** et **mod** ;
 - les opérateurs unaires - et +;
 - les opérateurs binaires - et +.



II.3.3.1 Type entier (opération)

■ Les expressions

- 2 est une expression réduite à une seule constante ;
- $3+4$ est une expression de valeur 7 ;
- -2 est une expression de valeur -2 ;
- **abs**(-2) est une expression de valeur 2 ;
- $2^{**}8$ est une expression de valeur 256 ;
- $5 / 2$ est une expression de valeur 2 ;
- $4 \text{ rem } 2$ et $4 \text{ mod } 2$ sont deux expression de valeur 0
- $5 \text{ rem } 2$ et $5 \text{ mod } 2$ sont deux expression de valeur 1
- $5 \text{ rem } (-2)$ est une expression de valeur 1
- $(-5) \text{ rem } 2$ est une expression de valeur -1
- $2+3*4$ est une expression qui vaut 14 ;
- $2+(3*4)$ est une expression qui vaut 14 ;
- $(2+3)*4$ est une expression qui vaut 20 .

■ Affectation

nom_de_variable := expression_de_type_Integer

procedure Exemple **is**

Max : constant := 5 ; *-- Une constante entière*

Nombre : integer ; *-- Une variable entière*

begin

Nombre := 5 ; *-- Affecte la valeur 5 à Nombre*

Nombre := Nombre +4 ; *-- Affecte la valeur 9 à Nombre*

Nombre := (36/10)*2 ; *-- Affecte la valeur 6 à Nombre*

Nombre := Max ; *-- Affecte la valeur Max à Nombre*

end Exemple;

II.3.3.1 Type entier (opération)

- Dangers liés aux variables non initialisées

procedure Exemple **is**

Nombre : integer ; -- *Deux variables entières sans valeur initiales définies*

Resultat : Integer ;

begin

Resultat := Nombre + 4 ; -- *Ajoute la valeur 4 à Nombre, et affecte le resultat a*
 -- *Resultat (résultat imprévisible)*

end Exemple;

II.3.3.1 Type entier (attributs)

- Integer'First; -- *donne le nombre le plus petit des entiers du type Integer*
- Integer'Last; -- *donne le nombre le plus grand des entiers du type Integer*
- Integer'Succ(0); -- *donne 1*
- Integer'Pred(0) ; -- *donne -1*
- Integer'Succ(Nombre) ; -- *donne la valeur Nombre + 1*
- Integer'Pred(Nombre+1); -- *donne la valeur Nombre*
- Integer'Max(Nombre1, Nombre2); -- *donne le plus grand des 2 nombres*
- Integer'Min(Nombre1, Nombre2); -- *donne le plus petit des 2 nombres*

II.3.3.1 Type entier (opération)

■ Type Short et Long

➤ la représentation machine dépend du compilateur

➤ Avec Integer sur 16 bits :

✓ Integer'First vaut $-2^{15} = -32768$;

✓ Integer'Last vaut $2^{15}-1 = +32767$

➤ Avec Short_Integer sur 8 bits : (Paquetage : Short_Integer_Text_IO)

✓ Short_Integer'First vaut $-2^7 = -128$

✓ Short_Integer'Last vaut $2^7-1 = +127$

➤ Avec Long_Integer sur 32 bits : (Paquetage : Long_Integer_Text_IO)

✓ Long_Integer'First vaut $-2^{31} = -2147483648$ (10 caractères)

✓ Long_Integer'Last vaut $2^{31}-1 = +2147483647$

III.3.3 Présentation d'un type énumération

- La présentation de type *énumération* est une liste ordonnée de valeurs distinctes représentée par des identificateurs et/ou des caractères littéraux ; cette liste est appelé énumération littérale.

- Exemple

(..., 'A', 'B', ...);	type prédéfini caractère
(False, True);	type prédéfini booléen
(Rouge, Orange, Vert)	type feux défini par l'utilisateur ;
(Lun, Mar, Mer, Jeu, Ven, Sam, Dim);	type jours de semaine
	défini par l'utilisateur

- Syntaxe pour définir un type énumération

type T_Piece **is** (Pile, Face) ; *-- pas de caractere accentue*

III.3.3 Utilisation d'un type énumération

```

procedure Exemple is
    type T_Fruits is (Orange, Banane, Pomme);
    type T_Legumes is (Carotte, Poireau);
    Un_Fruit   : T_Fruits ;
    Un_Legume  : T_Legumes ;

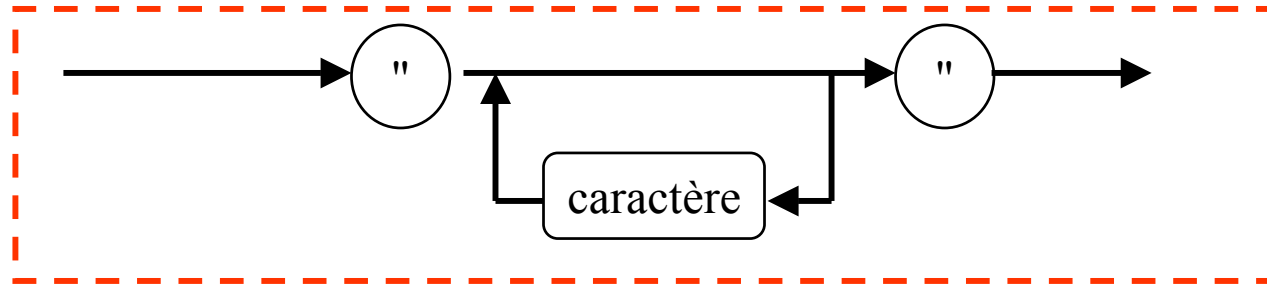
    procedure Eplucher (Fruit : T_Fruits) is
begin -- Eplucher
        null;
end Eplucher;

begin -- Exemple
    Un_Fruit   := Banane;
    Un_Legume  := Banane;           -- erreur !
    Eplucher (Un_Legume);         -- erreur !
end Exemple;

```

II.3.4 Chaîne de caractères, et caractères-littéraux

Une *chaîne de caractère* est définie par le diagramme suivant :



- la double apostrophe est représentée deux fois. Mais elle compte 1 seule fois dans le calcul de la longueur
- Une chaîne doit figurer entièrement sur une même ligne (dont la longueur est définie par l'implémentation
- Un caractère-littéral est un signe imprimable placé entre deux apostrophes simples

Exemple : "ABC"

"" chaîne vide

" " chaîne contenant un blanc

"Aa" 2 caractères différents

II.3.4.1 La table ASCII

- Pour Ada 83, la table est sur 7 bits seulement

Déci	Hexa	ASCII	Déci	Hexa	ASCII	Déci	Hexa	ASCII	Déci	Hexa	ASCII
64	40	@	80	50	P	96	60	`	112	70	p
65	41	A	81	51	Q	97	61	a	113	71	q
66	42	B	82	52	R	98	62	b	114	72	r
67	43	C	83	53	S	99	63	c	115	73	s
68	44	D	84	54	T	100	64	d	116	74	t
69	45	E	85	55	U	101	65	e	117	75	u
70	46	F	86	56	V	102	66	f	118	76	v
71	47	G	87	57	W	103	67	g	119	77	w
72	48	H	88	58	X	104	68	h	120	78	x
73	49	I	89	59	Y	105	69	i	121	79	y
74	4A	J	90	5A	Z	106	6A	j	122	7A	z
75	4B	K	91	5B	[107	6B	k	123	7B	{
76	4C	L	92	5C	\	108	6C	l	124	7C	
77	4D	M	93	5D]	109	6D	m	125	7D	}
78	4E	N	94	5E	^	110	6E	n	126	7E	~
79	4F	O	95	5F	_	111	6F	o	127	7F	DEL

II.3.4.2 Caractères (Attributs)

- Character'First donne le caractère de code 0 appelé *nul* ;
- Character'Last donne 'ÿ', le caractère de code 255 ;
- Chararcter'Pred('B') donne 'A' ;
- Chararcter'Succ('A') donne 'B' ;
- Chararcter'Pos('A') donne 65, le code de 'A' ;
- Chararcter'Val(65) donne 'A' ;
- Chararcter'Val(32) donne ' ', le caractere *espace* ;
- Chararcter'VA1(300) provoque une erreur à la compilation.
- Character'Min('A', 'F') donne le minimum entre A et F
- Character'Max('A', 'F') donne le maximum entre A et F
-A < B < C < D < a < b < c < caractères ordonnés

II.3.4.3 Traduction des opérations de lecture et d'affichage

Notation algorithmique

- **affectation :**
V ← expression

- **lecture :**
lire(X)

- **écriture :**
écrire("Total : ", T);
écrire("Total :",T,finligne);
écrire('a');

ADA

- **affectation :**
V := expression;

- **lecture :**
X := Get_Line; -- pour les chaînes
Get(X); -- pour les autres types

- **écriture :**
Put("Total : "); Put(T);
Put("Total : "); Put_Line(T);
Put('a');

III.3.3 Présentation de l'énumération prédéfini Boolean

Boolean est un **identificateur prédéfini** pour représenter un type énumération qui prend les valeurs *False* ou *True*

- Les opérations possibles les booléen sont : **and or xor** (binaire) **not**(unaire)
- Les opérateur qui renvoient un booléen : = /= <= < >= > in

Notation algorithmique

ADA

- **opérations sur les booléens :**
 - non**
 - et**
 - ou**
 - et puis ou alors**
- **comparaisons :**
 - = ≠
 - < >
 - ≤ ≥

- **opérateurs sur les booléens :**
 - not (priorité 1)**
 - and (priorité 2)**
 - or (priorité 3)**
 - ADA évalue toujours la totalité de l'expression**
- **comparaisons (priorité 4) :**
 - = /=
 - < >
 - <= >=

III.3.4 Exemple

procedure Volume_Travail **is**

type T_Jour **is** (Lun, Mar, Mer, Jeu, Ven, Sam, Dim);

Nb_Heures : Integer := 0;

begin

for Jour **in** T_Jour **loop** *-- for Jour in Lun..Dim loop*

If Jour **/=** Dim **then**

Nb_Heures := Nb_Heures + 2; *-- je travaille 2 heures par jours sauf le dimanche*

end if;

end loop;

end Volume_Travail ;

procedure Volume_Travail **is**

type T_Jour **is** (Lun, Mar, Mer, Jeu, Ven, Sam, Dim);

Nb_Heures : Integer := 0;

Jour : T_Jour;

begin

-- Jour := T_Jour'Val(0);

-- Jour := T_Jour'Val(T_Jour'Pos(T_Jour'First));

Jour := T_Jour'First;

while Jour **<** T_Jour'Last **loop**

Nb_Heures := Nb_Heures + 2;

Jour := T_Jour'Succ(Jour);

end loop;

end Volume_Travail ;

Expressions booléennes

II.3.3.2 Type réel (opération)

- les opérateurs arithmétiques
 - les opérateur unaires + - abs
 - les opérateurs binaires + - * / **

- les expressions réelles sont des combinaisons de ces opérations
 - 1.0 est une expression réduite à une seule constante de valeur 1.0 qui peut également s'écrire 1.0e0, 0.1e1, 0.1E+1, 10.0e-1 etc..
 - 2.0**(-8) est une expression de valeur 256^{-1} ;
 - -3.0+4.0 est une expression de valeur 1.0 ;
 - 4.3*5.0e0 est une expression de valeur 21.5 ;
 - 4.0 / 2.0 est une expression de valeur 2.0 ;
 - 5.0 / 2.0 est une expression de 2.5 (comparer avec l'exemple type entier) ;
 - 2.0+3.0*4.0 est une expression qui vaut 14.0 ;
 - 2.0+(3.0*4.0) est une expression qui vaut 14.0 (parenthèses inutiles) ;
 - (2.0+3.0)*4.0 est une expression qui vaut 20.0 .
 - abs (-2.0) donne 2.0, avec les parenthèses indispensables;
 - 3.0 * (-2.0) donne - 6.0, avec les parenthèses indispensables;
 - 5.0 ** (-2) donne 25.0^{-1} , avec les parenthèses indispensables.

II.3.3.2 Type réel (attributs)

- La représentation machine dépend du compilateur (mantisse et exposant)
- Affectation : Se fait de manière usuelle entre variable et une expression de même type

`nom_de_variable_réelle := expression_de_type_Float;`

- Les attributs utilisables sont (pour l'instant) et dépendent de l'environnement

First, Last, Digits, "Succ, Pred"

- les valeurs rendues sont de type :

Float pour First et Last

Integer pour Digits (nombre de chiffres significatifs exactes)

- Succ et Pred dépendent de la précision absolue du type

Toutefois, comme pour les entiers, une implémentation peut mettre à disposition d'autres types réels, par exemple:

Short_Float et Long_Float

Notation algorithmique

ADA

- opérations sur les entiers :

+ - * / div mod ^

exemple d'expression :

$A^8 + B * C \text{ div } D + E \text{ mod } 3$

- opérations sur les réels

+ - * /

pent pdec (partie entière et partie décimale)

pent(X)

pdec(X)



- opérateurs sur les entiers :

+ - * / / mod **

$A^{**8} + (B * C) / D + (E \text{ mod } 3)$

- opérateurs sur les réels

+ - * /



Integer(Float'truncation(X))



X - Float'truncation(X)

II.3.3.5 Conversions de types

Pour faire une conversion donnée => nom_de_type(expression)

- Float(5) 5 est converti en 5.0
- Float(-800)/2.5E2 l'expression vaut -3.2
- Float(-800/2.5E2) Erreur, mélange de types =>levée d'exception
- Integer(2.3) 2.3 est converti en 2
- Integer(3.5) 3.5 est converti en 4
- Integer(-2.5) -2.5 est converti en -3

Les valeurs entières max. et min. utilisables : `System.Min_Int .. System.Max_Int`

...

I : Integer := 34;

F : Float := 3.14;

C : Character := 'A';

begin

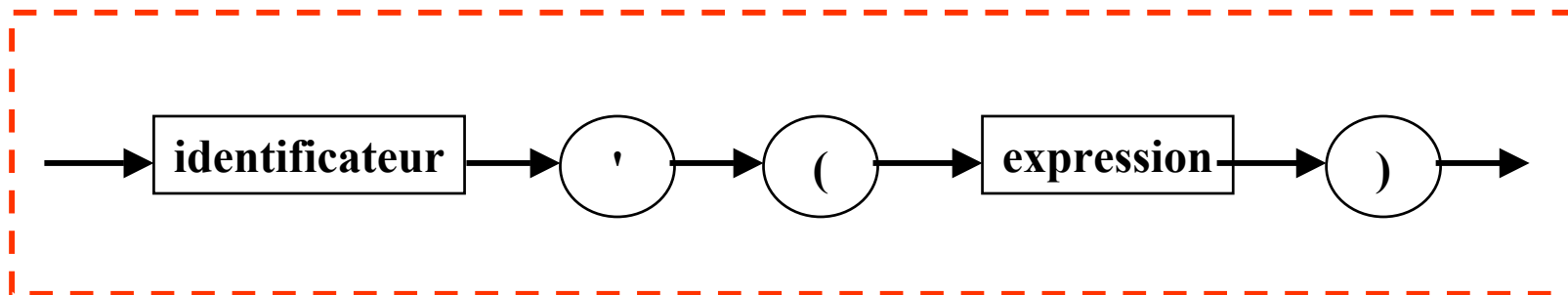
I := F; -- Cette instruction n'est pas valide

I := Integer(F); -- Il faut convertir le Float F en Integer

I := Integer(C); -- Cette instruction n'est pas valide, On ne peut pas convertir un
 -- caractère en un entier car ce sont deux objets de nature trop
 -- différente.

■ Les expressions

- Une expression **statique** : calculable à la **compilation**.
- Une expression **dynamique** : connue qu'à l'**exécution**.
- Une expression **qualifiée** consiste à préciser le type d'une expression.



Exemple d'expressions qualifiées (surtout utilisées lors de l'affichage)

Integer'(10)

le nombre 10 est ici du type Integer;

Long_Integer'(10)

le nombre 10 est ici du type Long_Integer;

Float'(10.0)

le nombre 10.0 est ici du type Float;

Short_Float'(10.0)

le nombre 10.0 est ici du type Short_Float;

III.3.7 Lecture et affichage des types prédéfinis

- Type Integer : *--type entier prédéfini*
with Ada.Integer_Text_IO; **use** Ada.Integer_Text_IO;

- Type Character : *--type caractère prédéfini*
with Ada.Text_IO; **use** Ada.Text_IO;

- Type Float *--type réel prédéfini*
with Ada.Float_Text_IO; **use** Ada.Float_Text_IO;

III.2 Contraintes

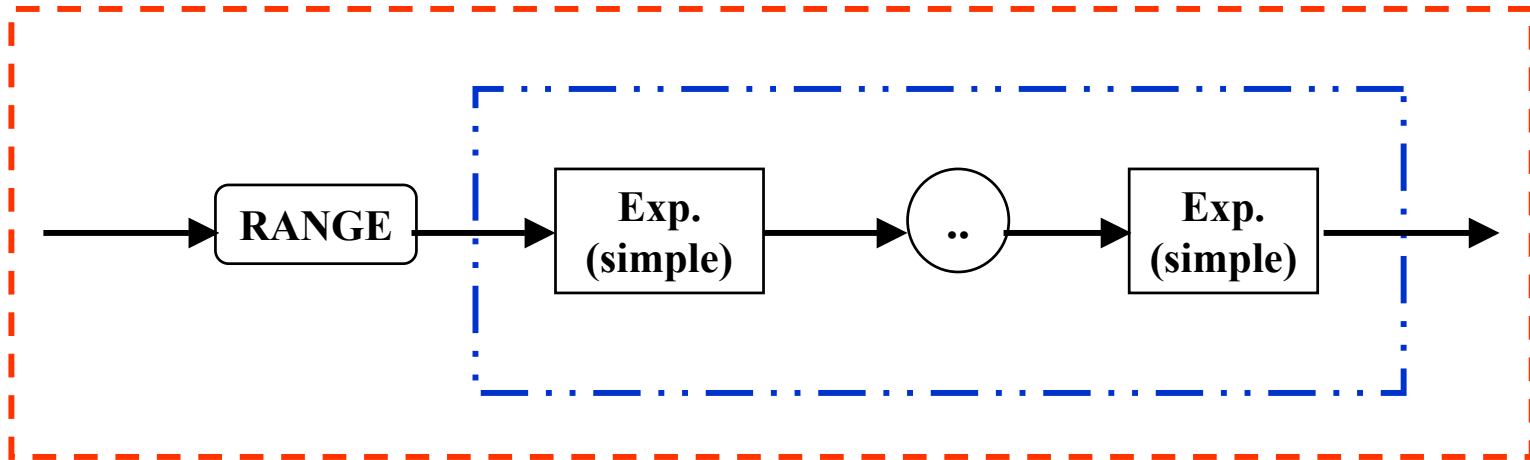
- *Une contrainte* est la restriction des valeurs possibles, définissant un sous ensemble muni des *mêmes opérations* que celles autorisées sur l'ensemble associé au type de base.

- Il existe 4 sortes de contraintes :
 - La contrainte de domaine, spécifiant des bornes inf et sup ; elle n'est applicable que à un type autorisant une relation d'ordre, c'est à dire associé :
 - ✓ à un ensemble discret (suite des valeurs ordonnées)
 - ✓ à l'ensemble des réels

 - La contrainte de précision
 - La contrainte d'indice
 - La contrainte de discriminant

III.2.1 Contrainte de domaine

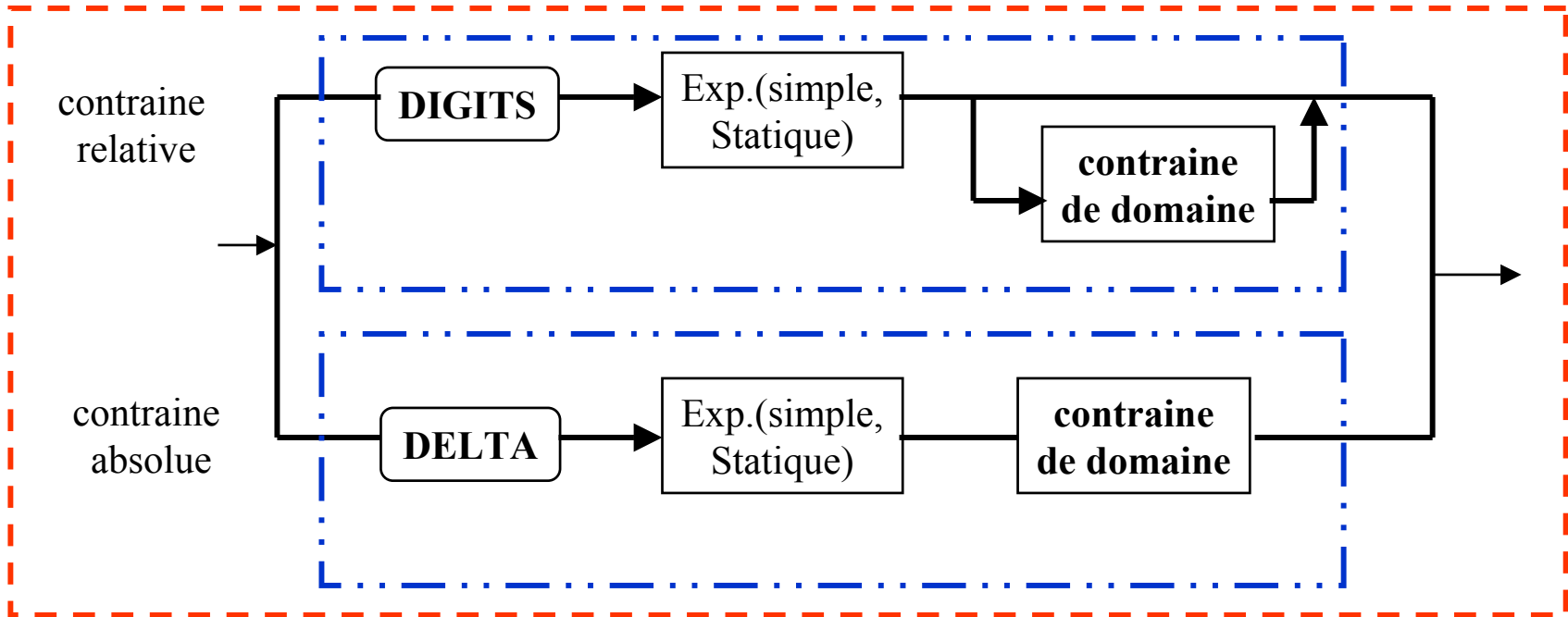
- Une *contrainte de domaine* est définie par le diagramme syntaxique suivant :



Exemple : **range** $-10..10$ (entier entre -10 à $+10$ inclus, 0 compris)
range 'A'..'Z' (lettres de l'alphabet)
range $-1.0..1.0$ (réel entre -1 et $+1$ inclus)

III.2.2 Contrainte de précision

- Une *contrainte de précision* est définie par le diagramme syntaxique suivant :



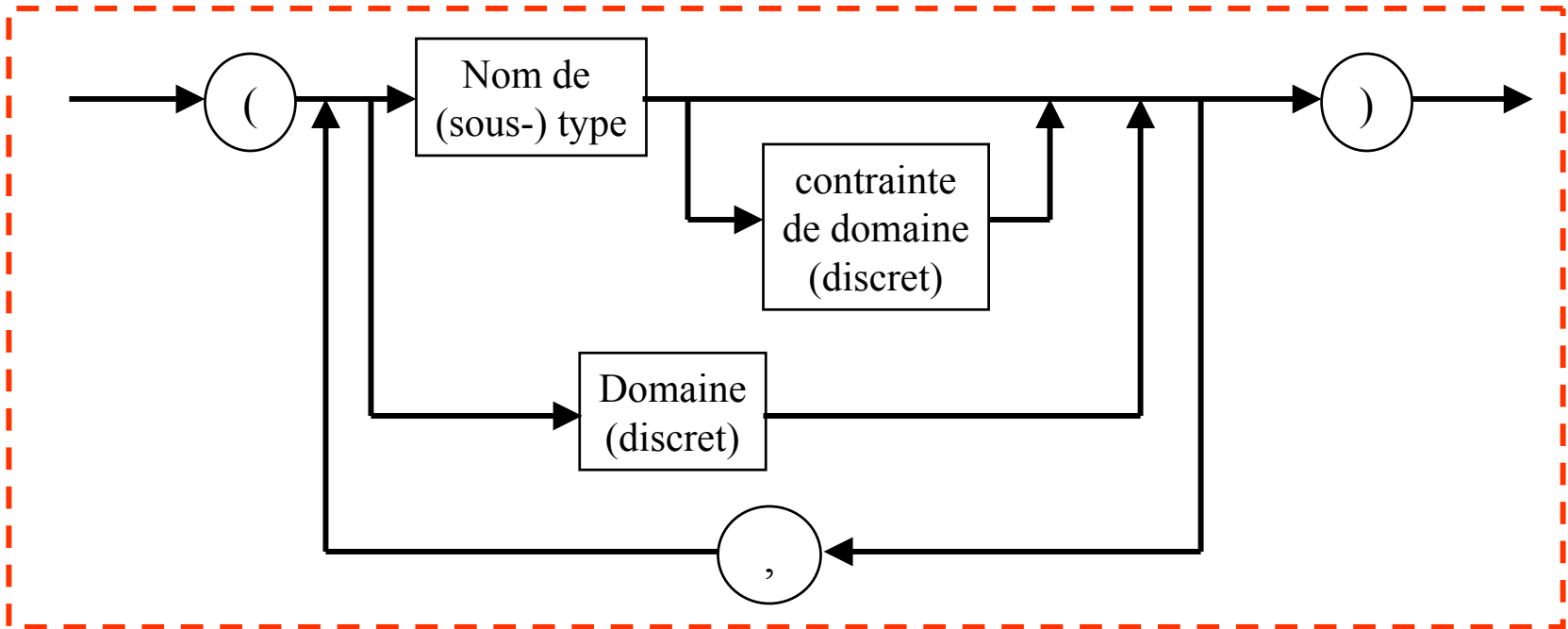
Exemple :

Digits 3 réel au moins à 3 chiffres
digits 3 range 20.0..50.0

Delta 0.1 réel avec pas d'un dixième
delta 0.1 range 35.0..40.0

III.2.3 Contrainte d'indice

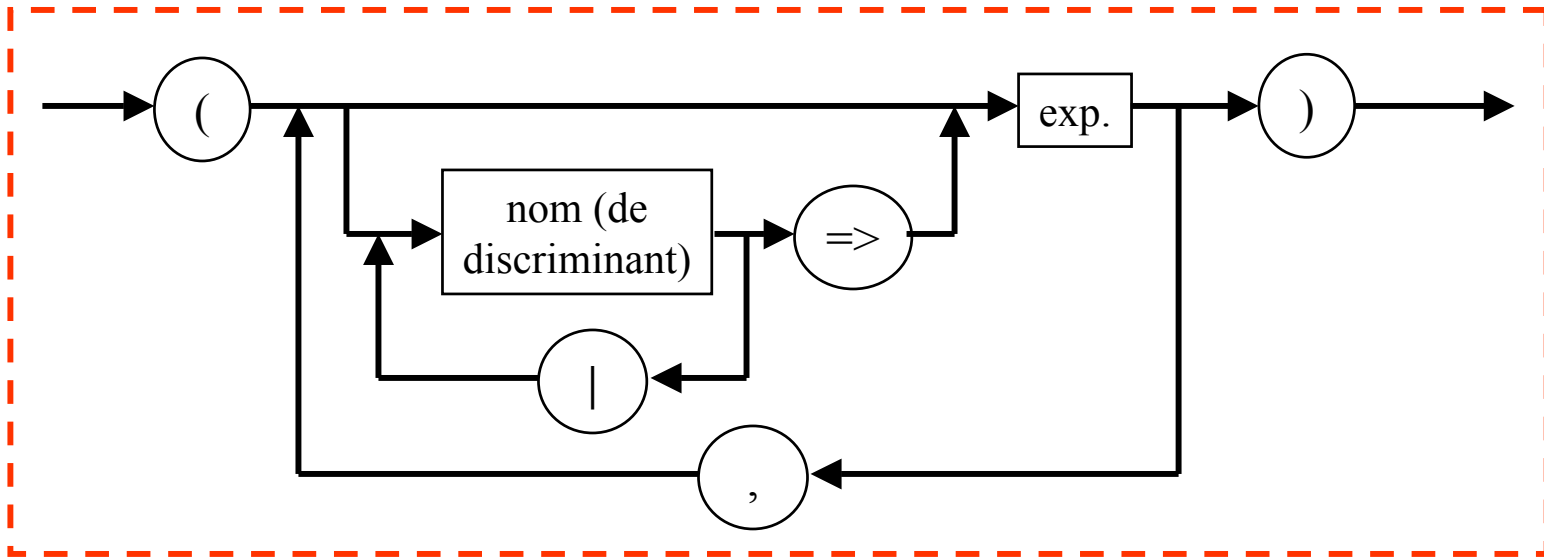
- Une *contrainte d'indice* est définie par le diagramme syntaxique suivant :



- Exemple :
- (Integer, Integer) permet de définir un tableau à 2 indices entiers
 - (Integer **range** -1..1, Integer **range** 1..3) permet de définir une matrice à 9 composantes
 - (1..5, 1..5) permet de définir une matrice carré à 25 composantes

III.2.4 Contrainte de discriminant

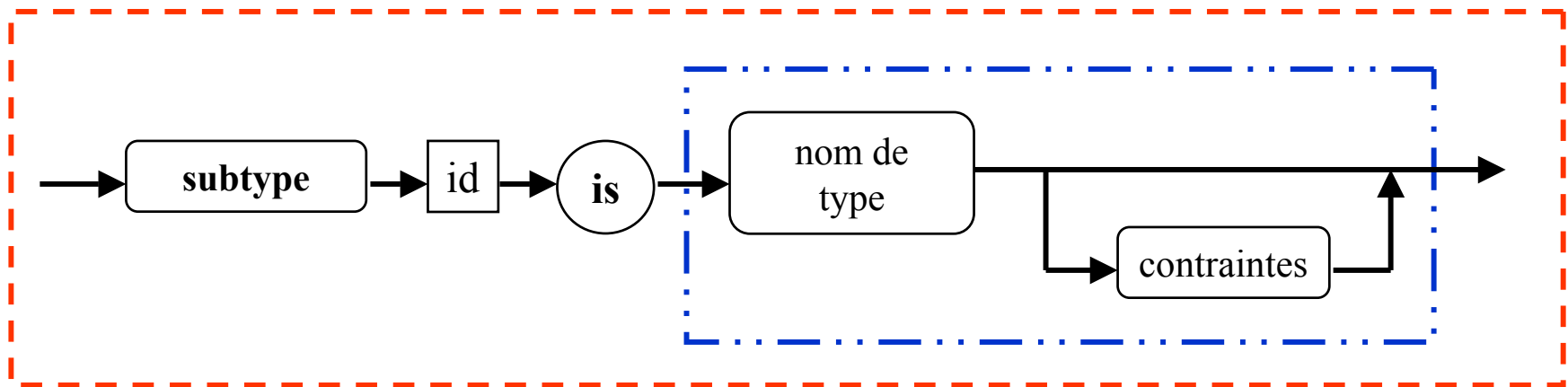
- Une *contrainte de discriminant* est définie par le diagramme syntaxique suivant :



Exemple :

(100) permettra de donner une valeur de 100 à un discriminant
 (SEXE => HOMME) associera une sorte d'enregistrement liée
 au discriminant SEXE

- Les sous-types : Sous-ensemble des valeurs d'un type



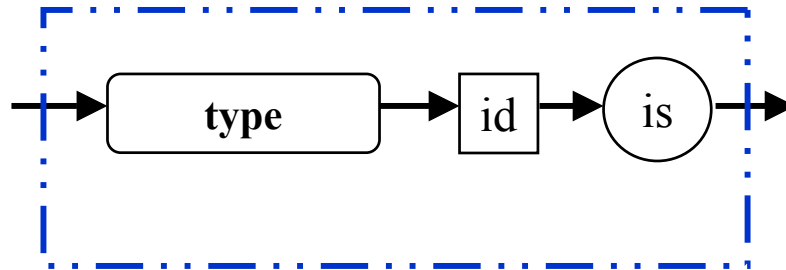
Cette déclaration donne l'identificateur du sous type et sa relation avec le type de base.

- Le sous-type est défini par :
 - une partie de l'ensemble de valeurs du type donné
 - l'ensemble des opérations primitives du type donné
- Le langage le considère comme le même type
- Pour les entiers, on peut citer les sous-types Natural et Positive
- Exemple d'un sous-type d'un type défini par l'utilisateur :

```
type T_heures is range 0 .. 23;
subtype T_matin is T_heures range 0 .. 11;
```

III.3.1 Déclaration d'un nouveau type

- La déclaration d'un nouveau *type* est définie par la diagramme suivant..



- La représentation d'un *nouveau type* revêt une des formes suivantes :
 - type énumération
 - type entier
 - type réel
 - type dérivé (juste une introduction)
 - type composé : les tableaux (à voir dans le chapitre tableau)
 - type composé : les articles (à voir dans le chapitre articles)
 - type accès (pointeur d'objets) (n'est pas au programme)
 - type tâches (processus en parallèle) (n'est pas au programme)

III.3.5 Présentation d'un nouveau type entier contraint

- La présentation d'un nouveau type *entier* est réduit à l'ajout d'une contrainte de domaine, dont les bornes doivent être entières et distinctes (la borne inférieure est strictement plus petite que la borne supérieure).

range min..max ; -- *notion d'ordre sinon type vide*

Exemple : A entier compris entre 1 et 12 -- *type entier signé*

Syntaxe :

type T_Douze **is new** Integer **range** 1..12 ;

type T_Douze **is range** 1..12 ; -- *new Integer est facultatif pour les entiers*

A : T_Douze :=1;

B : integer;

B := A; -- *affectation refusée par le compilateur*

package Douze_IO **is new** Ada.Text_IO. Integer_IO(T_Douze);

use Douze_IO;

III.3.5 Présentation d'un nouveau type énumération

- **Type** T_Jour **is** (Lun, Mar, Mer, Jeu, Ven, Sam, Dim);
package Jour_IO **is new** Enumeration_IO(T_Jour); **use** Jour_IO ;
- **Type** T_Eau **is** (Ocean, Mer, Lac);
package Eau_IO **is new** Enumaration_IO (T_Eau); **use** Eau_IO;

Put(Mer) -- *il y au une ambigüité pour le compilateur, quel paquetage utilisé?*
 -- *pour lever cette ambigüité, utiliser l'expression qualifiée*

Put(T_Jour'(Mer)); -- *utilisation du paquetage Jour_IO*

Put(T_Eau'(Mer)); -- *utilisation du paquetage Eau_IO*

NOTES

Ada.Text_IO.Put('A'); -- *affiche le caractère A*

package Char_IO **is new** Ada.Text_IO.Enumeration_IO(Character);

Char_IO.Put('A'); -- *affiche le caractère 'A', entre avec les apostrophes*

III.3.6 Présentation d'un type nouveau réel contraint

- La présentation d'un nouveau type *réel* est réduite à l'ajout d'une contrainte de précision relative ou absolue (donc assortie éventuellement d'une contrainte de domaine).
 - contrainte de précision relative => Type réel point flottant
 - contrainte de précision absolue => Type réel point fixe

Exemple : Digit 6 ; *-- type réel point flottant*
 Delta 0.01 Range -1.00..1.00 ; *-- type réel point fixe*

Syntaxe : **type** Identificateur **is digits** 3 ; *-- range -1.00..1.00*
type Identificateur **is delta** 0.01 **range** -1.00..1.00 ;
type Identificateur **is new** Float **range** -1.00..1.00 ;

Remarque : Dans ce troisième cas, "**is new** Float" est obligatoire

- Type dérivé d'un type

Reprend les caractéristiques du type dont il dérive ...

- ensemble de valeurs
- ensemble des primitives

... mais considéré comme comme un type différent.

```
type T_poids is new Float;
```

```
type T_longueur is new Float;
```


III.3.8 Lecture et affichage des nouveaux types réels

- Type réel point flottant : **type** T_Reel_Flottant **is digits** nbchiffres ;
Où nbchiffres (statique) représente la précision désirée. On peut rajouter une contrainte de domaine.
 - L'erreur (précision) est relative
 - elle est spécifiée en donnant le nombre minimal de chiffres significatifs désiré.

```
type T_Reel_Flottant is digits 6 range 0.0..0.999999; -- le 0 ne compte pas
package ES_R_Flottant is new Ada.Text_IO.Float_IO (T_Reel_Flottant );
use ES_R_Flottant ;
```

- Type réel point fixe : Ce type est utile pour travailler sur des nombres réels consécutifs séparés d'un pas fixe (erreur absolue).
 - L'erreur est absolue
 - elle est spécifiée en donnant l'écart entre deux valeurs immédiatement successives.

```
type T_Reel_Fixe is delta 0.1 range -1.0..1.0;
package ES_R_Fixe is new Ada.Text_IO.Fixed_IO (T_Reel_Fixe);
use ES_R_Fixe ;
```

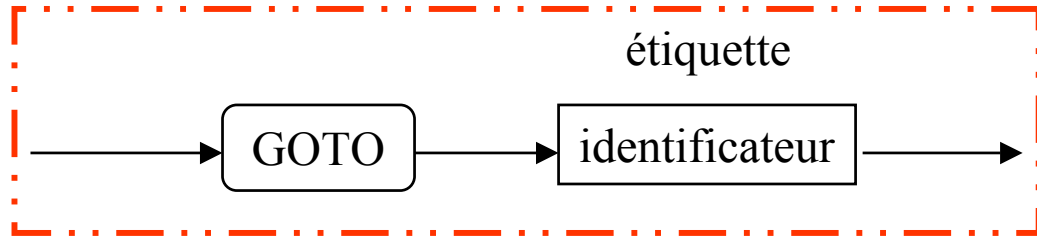
Sommaire

- Chapitre I : Présentation
- Chapitre II : Unités lexicales
- Chapitre III : Types et sous types
- **Chapitre IV : Ordres (Sélection, cas, itération, ...)**
- Chapitre V : Sous programmes
- Chapitre VI : Tableaux (array)
- Chapitre VII : Chaînes de caractères (String)
- Chapitre VIII : Articles (record)
- Chapitre IX : Pointeur
- Chapitre X : Fichiers (File)
- Chapitre XI : Paquetages simples (package)
- Chapitre XII : Généricité
- Chapitre XIII : Tâches

Un *ordre* est un constituant de programme qui entraîne *l'exécution d'actions*, par opposition aux déclarations qui élaboraient des types ou des objets et aux expressions qui évaluaient des résultats.

- IV.1 Affectation
- IV.2 Branchement
- IV.3 Alternative
- IV.4 Cas
- IV.5 Boucle
- IV.6 Bloc
- IV.7 Exceptions

- Le *branchement* est défini ainsi :



Exemple **begin**

if Y = 0 **then**

GOTO Label;

end if;

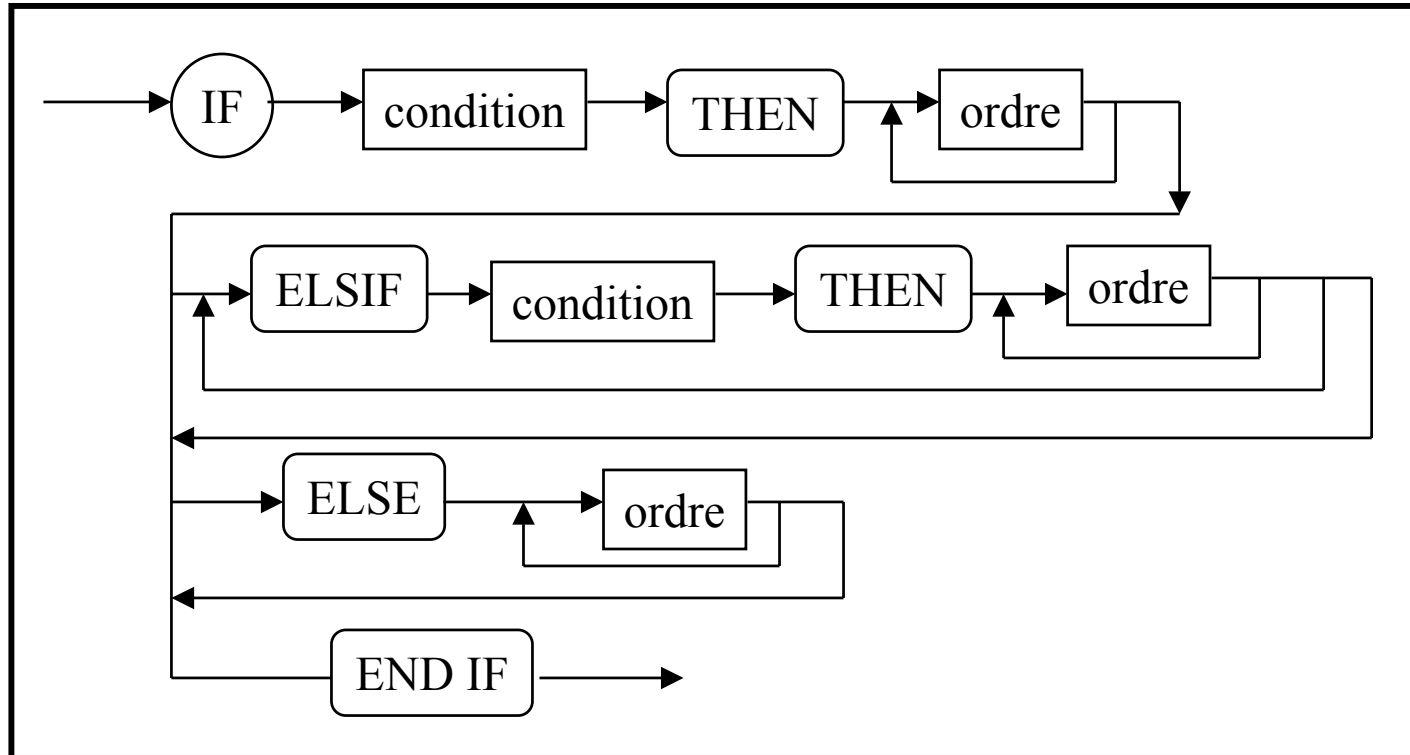
D := X/Y ;

<< Label >> D := Max ; --étiquette figurant ailleurs (lourde et laide)

end ;

- IL cause l'exécution de l'ordre désigné par l'étiquette et rompt donc le déroulement normal du programme qui est d'exécuter les ordres séquentiellement.
- IL n'est pas possible de l'utiliser de l'extérieur d'un ordre composé pour y rentrer, ni pour sauter d'une suite d'ordres à une autre dans un ordre composé.

L'*alternative* est un ordre composé défini ainsi :



■ L'alternative sert à choisir quel ordre, d'une suite, va être exécuté, suivant les valeurs de conditions.

Traduction du conditionnelle simple

Notation algorithmique

```

si condition
alors actions
fin si

```

```

si Ma_Lettre = 'A'
alors NB_A ← NB_A + 1
fin si

```

ADA

```

if condition then
    actions ;
end if;

```

```

if Ma_Lettre = 'A' then
    NB_A := NB_A + 1;
end if;

```

Traduction du conditionnelle avec une alternative

Notation algorithmique

```

si condition
alors action1
sinon action2
fin si
  
```

Exemple :

```

si  $X < Y$ 
alors  $MAX \leftarrow Y$ 
sinon  $MAX \leftarrow X$ 
fin si
  
```

ADA

```

if condition then
    action1;
else
    action2;
end if;
  
```

Exemple :

```

if  $X < Y$  then
     $MAX := Y;$ 
else
     $MAX := X;$ 
end if;
  
```

Traduction du conditionnelle avec plusieurs alternatives ou généralisée

Notation algorithmique

```
selon conditions  
  condition1 : action1  
  condition2 : action2  
  condition3 : action3  
fin selon
```

ADA

```
-- selon variables  
if condition1 then  
  action1;  
elsif condition 2 then  
  action2;  
elsif condition 3 then  
  action3;  
end if;
```


Exemple

Notation algorithmique

selon Condition sur X, Y

$X < Y : M \leftarrow Y$

$X = Y : M \leftarrow Y+X$

$X > Y : M \leftarrow X-Y$

fin selon

ADA

-- selon X, Y

if $X < Y$ then

$M := Y;$

elsif $X = Y$ then

$M := Y+X;$

elsif $X > Y$ then

$M := X-Y;$

end if

Traduction du conditionnelle avec plusieurs alternatives + un cas par défaut

Notation algorithmique

```
selon conditions  
  condition1 : action1  
  condition2 : action2  
  autrement : action3  
fin selon
```

ADA

```
-- selon variables  
if condition1 then  
  action1;  
else if condition 2 then  
  action2;  
else  
  action3;  
end if;
```

Exemple

Notation algorithmique

selon condition sur X, Y, Z

$X < Y : M \leftarrow Y$

$(X = Z)$ et $(Z > Y)$:

$M \leftarrow Y+X$

autre cas :

$M \leftarrow X-Y$

fin selon

ADA

-- selon X, Y

if $X < Y$ then

$M := Y;$

else if $X=Y$ and $Z>Y$ then

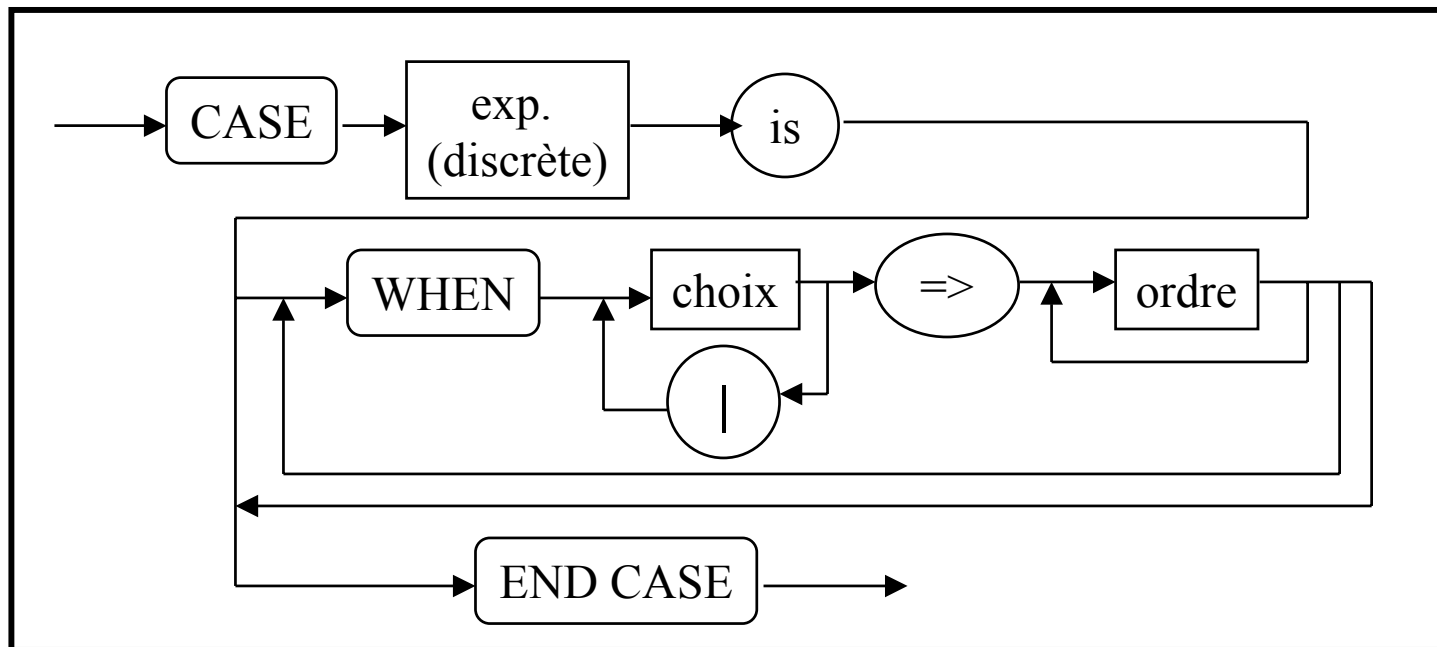
$M := Y+X;$

else

$M := X-Y;$

end if

Le cas est un ordre composé défini ainsi :



- Le choix a un rôle analogue à l'alternative, mais c'est l'égalité d'une valeur d'expression de type discret et d'un choix proposé qui détermine la suite des ordres à exécuter.
- Les choix doivent proposer toutes les valeurs possibles une fois et une seules fois, OTHERS servant à proposer toutes les restantes et ne pouvant donc être placé qu'en dernier.

Cas particulier d'analyse par cas

Notation algorithmique

```

selon V
  V = a : action1
  V = b : action2
  V = c : action3
fin selon
  
```

ADA

```

case V is
  when a => action1;
  when b => action2;
  when c => action3;
  when others => null;
end case;

Attention :
Il faut énumérer tous les cas.
Une action peut être l'instruction vide null
  
```

Espression d'un type discret

Statique, valeurs ou intervalles séparés par des barres verticales

Exemple

Notation algorithmique

selon V

V = 'a' : B ← 1

V = 'b' : B ← X

V = 'c' : B ← 3

V = 'd' : B ← 1

fin selon

ADA

case V is

when 'a' => B:=1;

when 'b' => B:=X;

when 'c' => B:=3;

when 'd' => B:=1;

when others => null;

end case;

Notation algorithmique

```

selon V
  V = a :      action1
  V = b :      action2
  V = c :      action3
  ...
  autrement : action-k
fin selon
  
```

ADA

```

case V is
  when a => action1;
  when b => action2;
  when c => action3;
  ...
  when others => action-k;
end case;
  
```

Exemple

Notation algorithmique

selon V

V = 'a' : B ← 1

V = 'b' : B ← X

C ← V

V = 'c' : B ← X

V = 'd' : B ← X

autrecas : C ← V

fin selon

ADA

case V **is**

when 'a' => B:=1;

when 'b' => B:=X;

C:=V;

when 'c' | 'd' => B:=X;

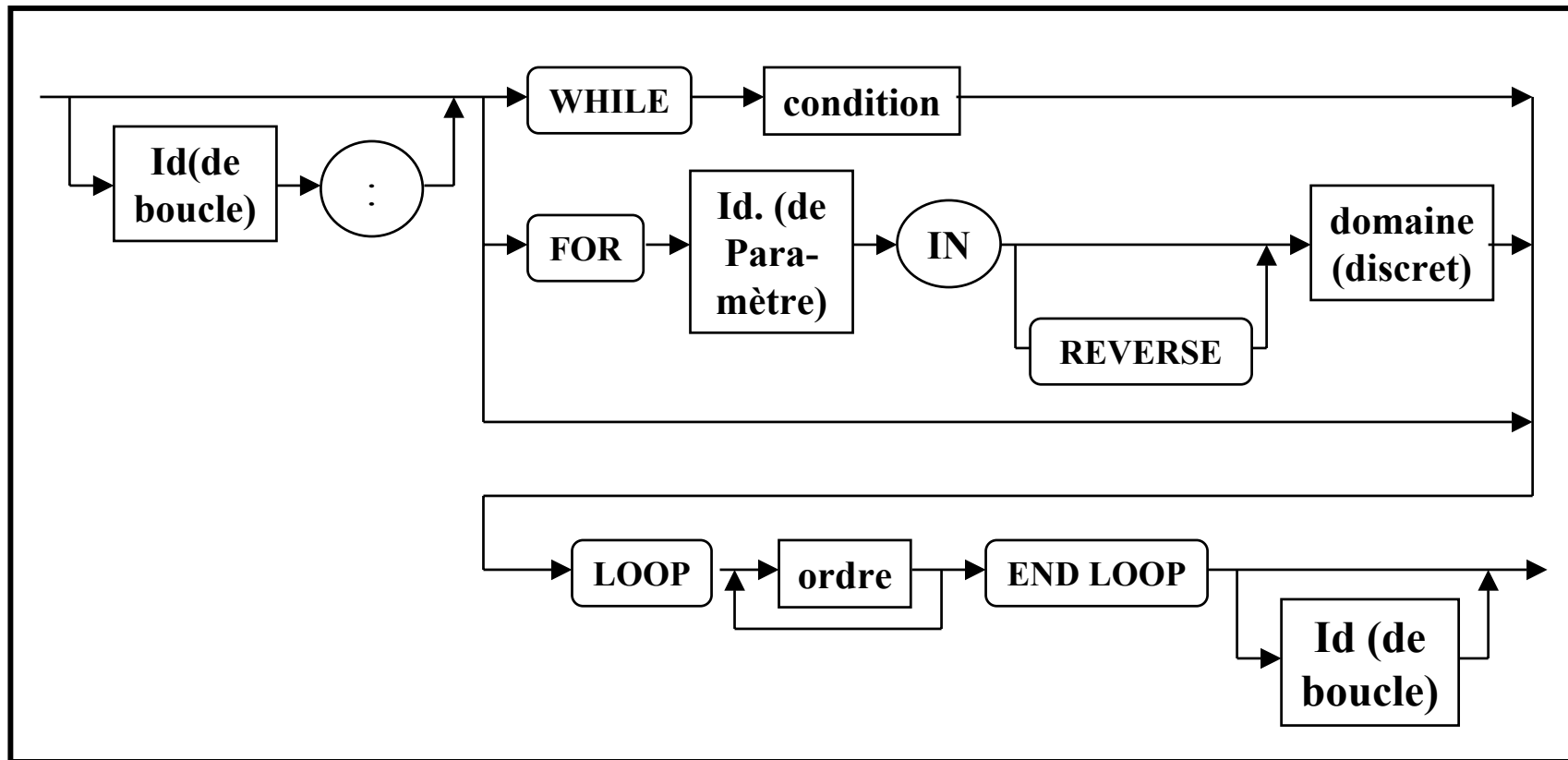
when 'e' ... 'k' | 's' => B:=0;

when others => C:=V;

end case;

- Introduction
- Boucle while
- Boucle for
- Boucle générale
- Sortie de boucle
- Exemple boucles imbriquées

- La *boucle* est un ordre composé défini ainsi :



- Une boucle permet d'exécuter une suite d'ordres un certain nombre de fois.

Boucle while

Notation algorithmique

```

Ident : tantque Ccont faire
        action
        ftq
N ← 10
i ← 1
S ← 0
Somme : tantque i ≤ N faire
        S ← S + i
        i ← i + 1
        fin tantque
    
```

ADA

```

Ident : while Ccont loop
        action;
        end loop;
N := 10;
i := 1;
S := 0;
Somme : while i <= N loop
        S := S + i ;
        i := i + 1;
        end loop Somme;
    
```

Remarque :

- Il faut s'assurer que l'expression booléenne devient fausse après un nombre fini d'itérations, sinon le programme exécuterait l'instruction **while** indéfiniment.
- Si l'expression booléenne est initialement fausse, l'instruction **while** n'est pas effectué

Boucle for

Notation algorithmique

ADA

répéter N fois

action

fin répéter

répéter pour I allant de 4 à 1 (4 fois)

AV(50+I)

GA(90+2*I)

fin répéter

for I in 1..N **loop**

action;

end loop;

for I in **Reverse** 1..4 **loop**

AV(50 + I);

GA(90+ 2 * I);

end loop;

- Le nombre d'itérations sera égal à l'expression₂ – expression₁ + 1
- La variable de boucle (variable de contrôle) est déclarée implicitement, du type des bornes de l'intervalle et n'existe que dans le corps de la boucle **for**
- Il n'est pas possible de changer la valeur de la variable de boucle
- Si l'intervalle est nul, c'est-à-dire que l'expression₁ a une valeur supérieure à expression₂, la boucle n'est pas effectuée
- Si la valeur de expression₁ ou expression₂ est modifiée par une itération, le nombre d'itérations ne change pas!

La boucle générale loop

Notation algorithmique

répéter

action

jusqu'à CondArrêt

$N \leftarrow 10$

$i \leftarrow 1$

$S \leftarrow 0$

répéter

$S \leftarrow S + i$

$i \leftarrow i + 1$

jusqu'à $i > N$

ADA

loop

action;

if CondArret **then exit;**

end if;

end loop;

$N := 10;$

$i := 1;$

$S := 0;$

loop

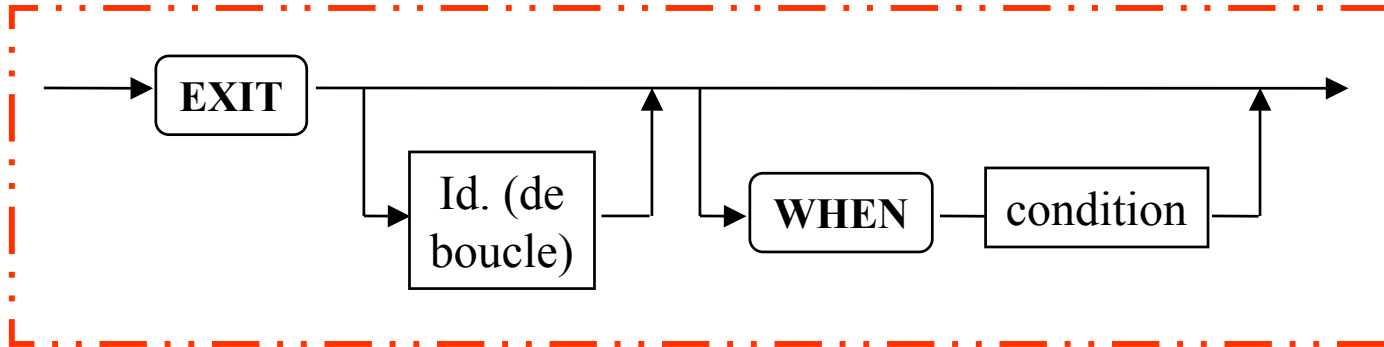
$S := S + i ;$

$i := i + 1;$

exit when $i > N;$

end loop;

- L'ordre de *sortie de boucle* est un ordre simple défini ainsi :



- IL cause la sortie d'une *boucle*, c'est à dire l'exécution du premier ordre suivant :
- ✓ Si aucun identificateur de boucle n'est mentionné, la boucle qui cesse d'être exécuté est celle englobant immédiatement l'ordre de sortie.
- ✓ Si plusieurs boucles sont imbriquées les unes dans les autres, la mention de l'identificateur permet de sortir simultanément de plusieurs boucles.

Exemple boucles imbriquées

S:=0.0;

N:= 1;

i := 0

Somme : **while** i <= N **loop**

 S := S + i;

 i := i + 1;

 SIGMA : **loop**

 U:= 1.0/REAL(N**2);

 S:= S+U;

exit Somme **when** U/S <1.0E-4 ;

 N:=N+1

end loop SIGMA;

end loop Somme;

Exemple boucles imbriquées

- Il est possible, mais déconseillé, de placer une ou plusieurs instructions **exit** dans une boucle **for** ou **while**.
- La boucle **loop** peut porter une étiquette. Cette étiquette est utile dans des cas tels que deux boucles imbriquées:

Externe: **loop**

loop

-- Instructions

exit when B; -- Sortie de la boucle interne si B vraie

-- Instructions

exit Externe when C; -- Sortie de la boucle Externe si C vraie

-- Instructions

end loop;

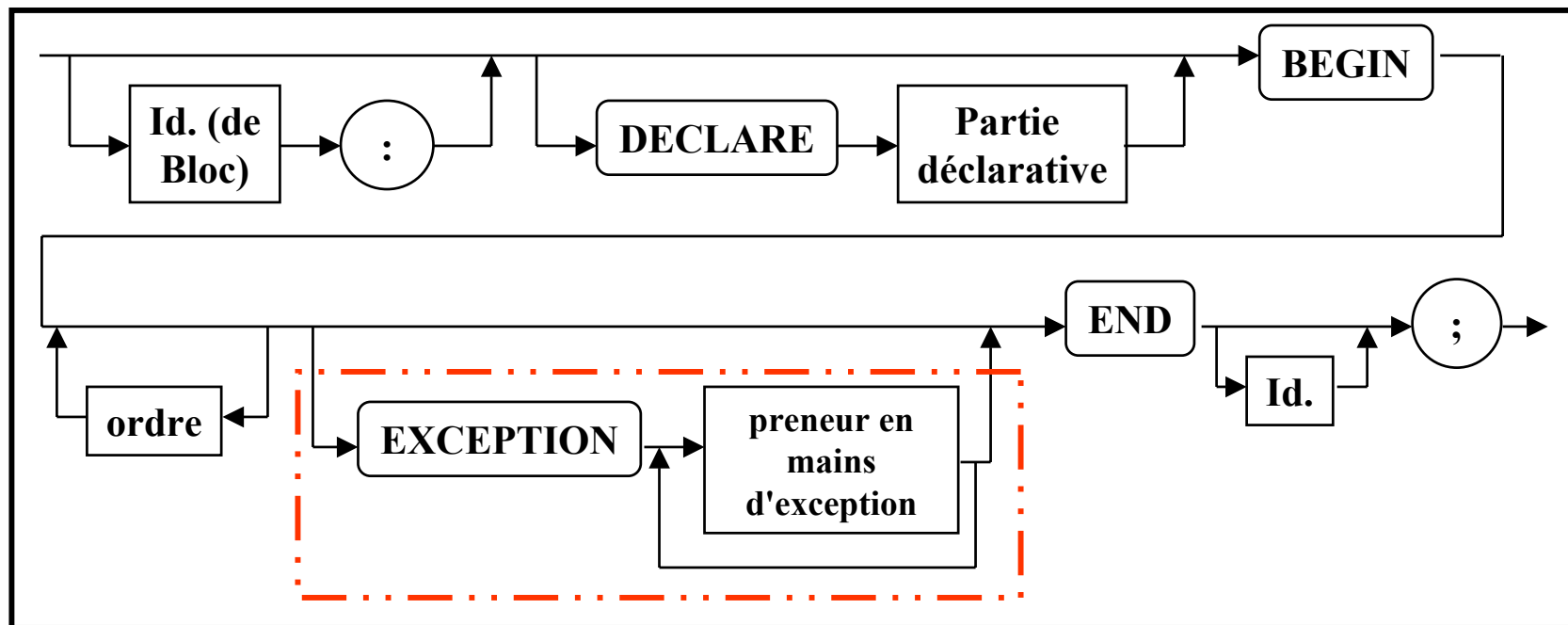
-- Instructions

end loop Externe;

- Les boucles **for** et **while** peuvent aussi porter une étiquette.
- L'instruction **exit;** sans condition existe et provoque la sortie inconditionnelle de la boucle.

IV.6 Bloc

Un *bloc* est un ordre composé, défini par le diagramme syntaxique suivant :



Exemple :

ECHANGE :

declare

Val_1 : Integer ; --*Val_1 a une existence uniquement*

begin

-- *à l'intérieur du bloc*

Val_1 := 10 ; -- *ordre d'affectation*

B := Val_1;

end ECHANGE ;

Portée

L'objet est potentiellement visible (l'objet existe, on parle aussi de durée de vie).

Visibilité.

On peut utiliser son identificateur pour y référer. Même si l'objet à une portée dans une région, il n'est pas obligatoirement visible. Il peut être masqué par un autre identificateur défini dans cette région

IV.6.2 Portée et visibilité

declare

I:Integer:=5;

begin

I:=I+1;

declare

K:Integer:=I;

I:Integer:=0;

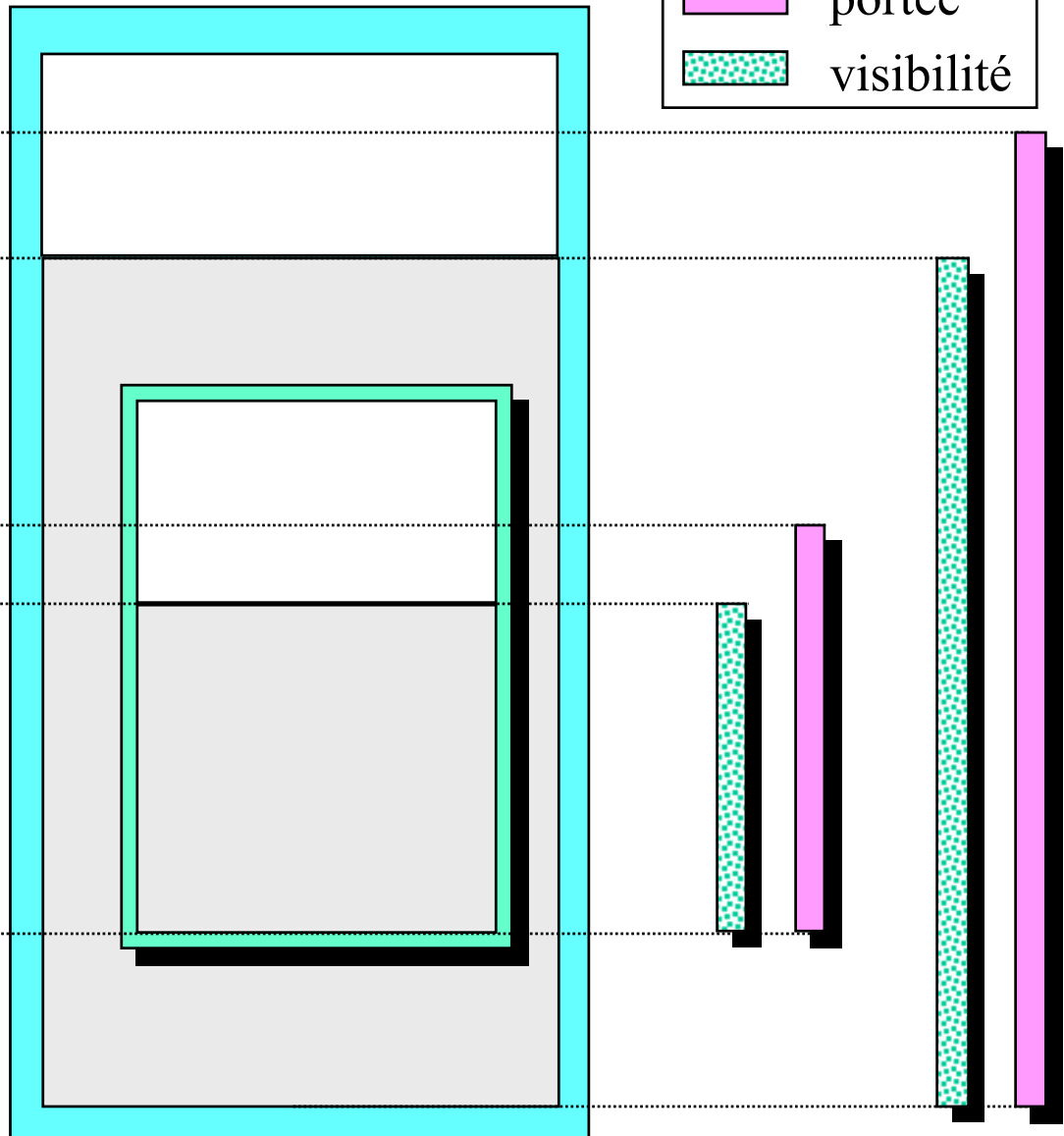
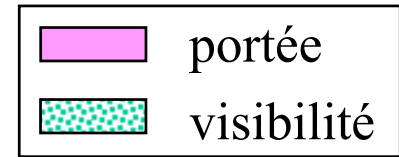
begin

K:=I;

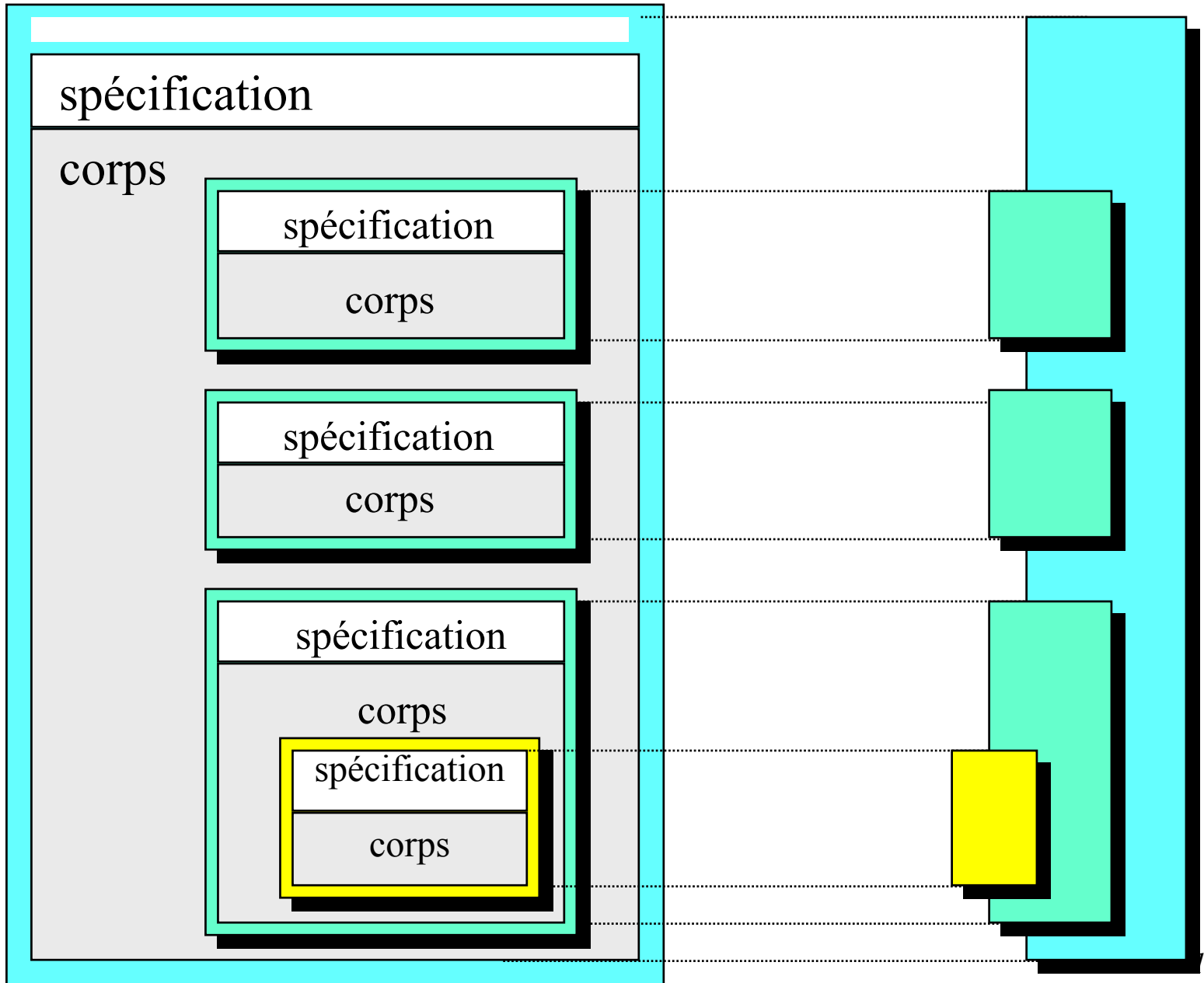
...

end

end



IV.6.2 Portée et visibilité



Exceptions

- Une exception indique une erreur pendant l'exécution
- Une *exception* est un événement, qui entraîne une suspension définitive de l'élaboration d'une déclaration ou de l'exécution d'un ordre. *A la place*, un *preneur en mains d'exception* est exécuté.
- Une exception est elle-même suscitée soit fortuitement (en particulier à cause d'une erreur), soit par un ordre donné par le programmeur
- => Utiliser les exceptions vous donnera un code plus sûr

```

with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Saisir is
    N : Positive;
begin
    Get(N);
    Skip_Line;
end Saisir;
  
```

Les exceptions

- Déclaration des exceptions
 - Exceptions prédéfinies
 - Exceptions déclarées par l'utilisateur

- Levée d'une exception
 - Automatique
 - Par l'utilisateur

- Exception prédéfinies ADA 95
 - `Constraint_Error` (violation de tout type de contrainte (domaine, précision, indice, discriminant, division par zéro))
 - `Data_Error` (données qui ne sont pas de même type)
 - `Programm_Error` (violation d'une structure de contrôle, arrivée sur le end d'une fonction)
 - `Storage_Error` (dépassement de la mémoire)
 - `Tasking_Error` (traitement des tâches, parallélisme)
 - `Numeric_Error` (division par zéro dans ADA 83)

- Traiter une exception

begin

Propagation d'une exception

begin

begin

begin

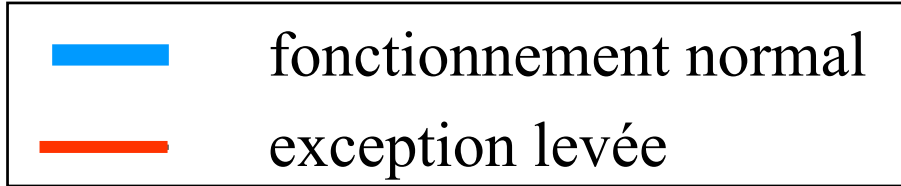


end

end

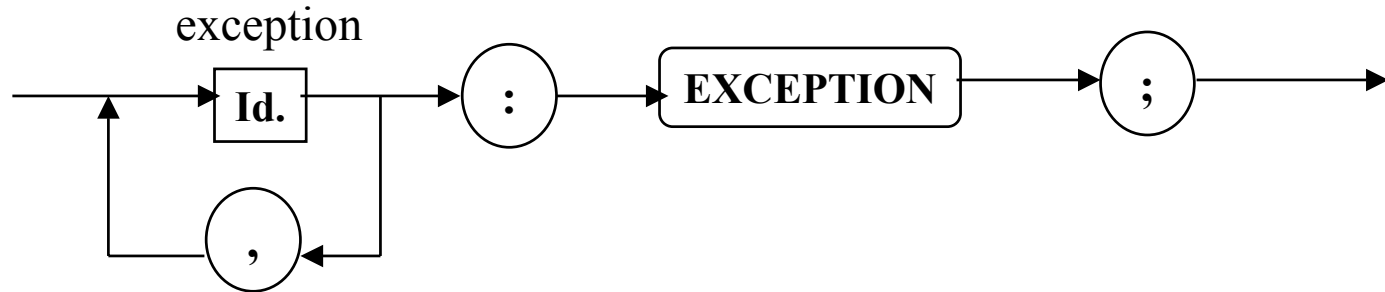
end

end



Déclaration et levée d'exception

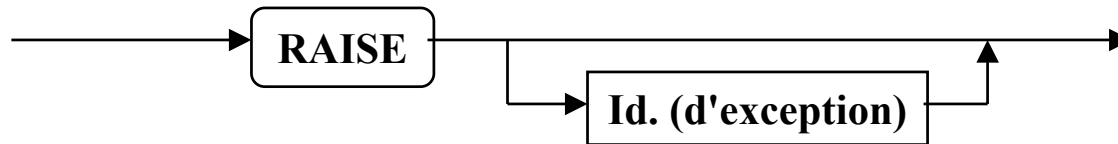
- Pour être reconnue comme telle, une exception doit avoir été préalablement déclarée selon le diagramme suivant :



Exemple :

DENOMINATEUR_NUL, DISCRIMINANT_NEGATIF : EXCEPTION ;

- Une exception peut être suscitée par un *ordre*, selon le diagramme suivant :



Exemple : **if** Delta < 0.0 **then raise** DISCRIMINANT_NEGATIF **end if;**

Remarque : L'*ordre raise* sans indication d'un identificateur d'exceptions sert uniquement à un preneur en mains d'exceptions.

- Le bloc de traitement d'exception est donné par le diagramme ci-dessous

```
Discriminant_Negatif : Exception ; -- declarer une nouvelle exception
```

```
exception -- sinon traitement des exceptions
```

```
    when Constraint_Error => Put_Line (" violation d'une contraine ");
```

```
    when Data_Error => Put_Line (" mauvaise type pour la saisie de Val ");
```

```
    when Discriminant_Negatif => Put_Line(" delta negatif ");
```

```
    when others => Put_Line(" erreur inconnue ");
```

Remarque : évitez **when others => null;** -- masquer l'erreur

Traitement d'une exception (exemple 1)

- **Lorsqu'une exception est levée, on peut intercepter cette exception, et faire un traitement.**

procedure Calcul **is**

Discriminant_Negatif : exception ; *-- déclarer une nouvelle exception*

subtype Mon_Reel **is** Float **range** -6.0..6.0; *-- pas de package d'entree/sortie, c'est un sous-type*

Val : Mon_Reel ;

Y, Delta : Float ;

begin *-- Calcul*

Get(Val);

Skip_Line;

Delta := Val;

if Delta < 0.0 **then raise** Discriminant_Negatif **end if**;

Y:= sqrt(Delta);

exception *-- sinon traitement des exceptions*

when Constraint_Error => Skip_Line;

Put_Line (" Mauvaise domaine de la saisie de Val ");

when Data_Error => Skip_Line;

Put_Line (" Mauvaise type pour la saisie de Val ");

when Discriminant_Negatif => Put_Line(" delta negatif ");

end Calcul;

Remarque : évitez **when others => null**;

Récupération d'une exception dans un sous programme et propagation vers le programme appelant

procedure Exemple is

A : Positive;

Erreur_Saisie : **exception** ;

procedure Saisir (Val : **out** Positive) is

begin -- *Saisir*

Put ("entrer la valeur ");

Get (Val); skip_line ;

exception

when Constraint_Error =>

Put_Line (" Débordement de type lors de la saisie de Val "); Skip_Line;

raise Erreur_Saisie ;

when Data_Error =>

Put_Line (" Mauvaise type pour la saisie de Val "); Skip_Line;

raise Erreur_Saisie;

end Saisir;

begin -- *Exemple*

Saisir (A) ; -- *appel de la procedure Saisir*

exception

when Erreur_Saisie => Put_Line (" Probleme dans la procedure de saisie des donnees ");

end Exemple;



Exception non traitée : le programme continue avec des données erronées

Traitement d'une exception (exemple 2)

- Contrôle de la saisie au clavier d'une variable d'un type donnée. Si la saisie est :
 - ✓ un caractère, alors l'exception levée est de type `Data_Error`
 - ✓ un nombre négatif ou nul, alors l'exception levée est de type `Constraint_Error`
 - ✓ un entier positif, la saisie est correcte, on quitte cette procédure avec la bonne valeur lue de `Val`.

procedure Saisir (Val : **out** Positive) **is**

begin

loop

begin

 Get(Val);

 Skip_Line;

exit; -- ou **return** Val si c'est une fonction qui retourne la valeur lue Val

exception -- sinon traitement des exceptions

when Constraint_Error => Skip_Line;

 Put_Line (" Débordement de type lors de la saisie de Val ");

 Put_Line(" Refaire la saisie... ");

when Data_Error => Skip_Line;

 Put_Line (" Mauvaise type pour la saisie de Val ");

 Put_Line(" Refaire la saisie... ");

end;

end loop;

end Saisir;

 Le sous programme fournit toujours une valeur entière positive

Sommaire

- Chapitre I : Présentation
- Chapitre II : Unités lexicales
- Chapitre III : Types et sous types
- Chapitre IV : Ordres (Sélection, cas, itération, ...)
- **Chapitre V : Sous programmes**
- Chapitre VI : Tableaux (array)
- Chapitre VII : Chaînes de caractères (String)
- Chapitre VIII : Articles (record)
- Chapitre IX : Pointeur
- Chapitre X : Fichiers (File)
- Chapitre XI : Paquetages simples (package)
- Chapitre XII : Généricité
- Chapitre XIII : Tâches

Chapitre V : Sous programmes

- Exemple de sous programme
- Déclaration de paramètres
- Traduction des paramètres formels
- Déclaration de sous programme
- Corps (body) d'un sous programme
- Appel de sous programme
- Arguments d'appel
- Portée des paramètres (Global, Local)
- Les effets de bords
- Surcharge des opérateurs

Exemple de sous programme

- Si une fonction ou une procédure a plusieurs paramètres formels, ils sont séparés par un ";"

- **procedure** Ma_Procedure(X : **in out** Integer ; Y : **in out** Float) ;

- On peut mettre en facteur le type des paramètres

procedure Permute(X, Y : **in out** Integer) ;

Paramètres
formels

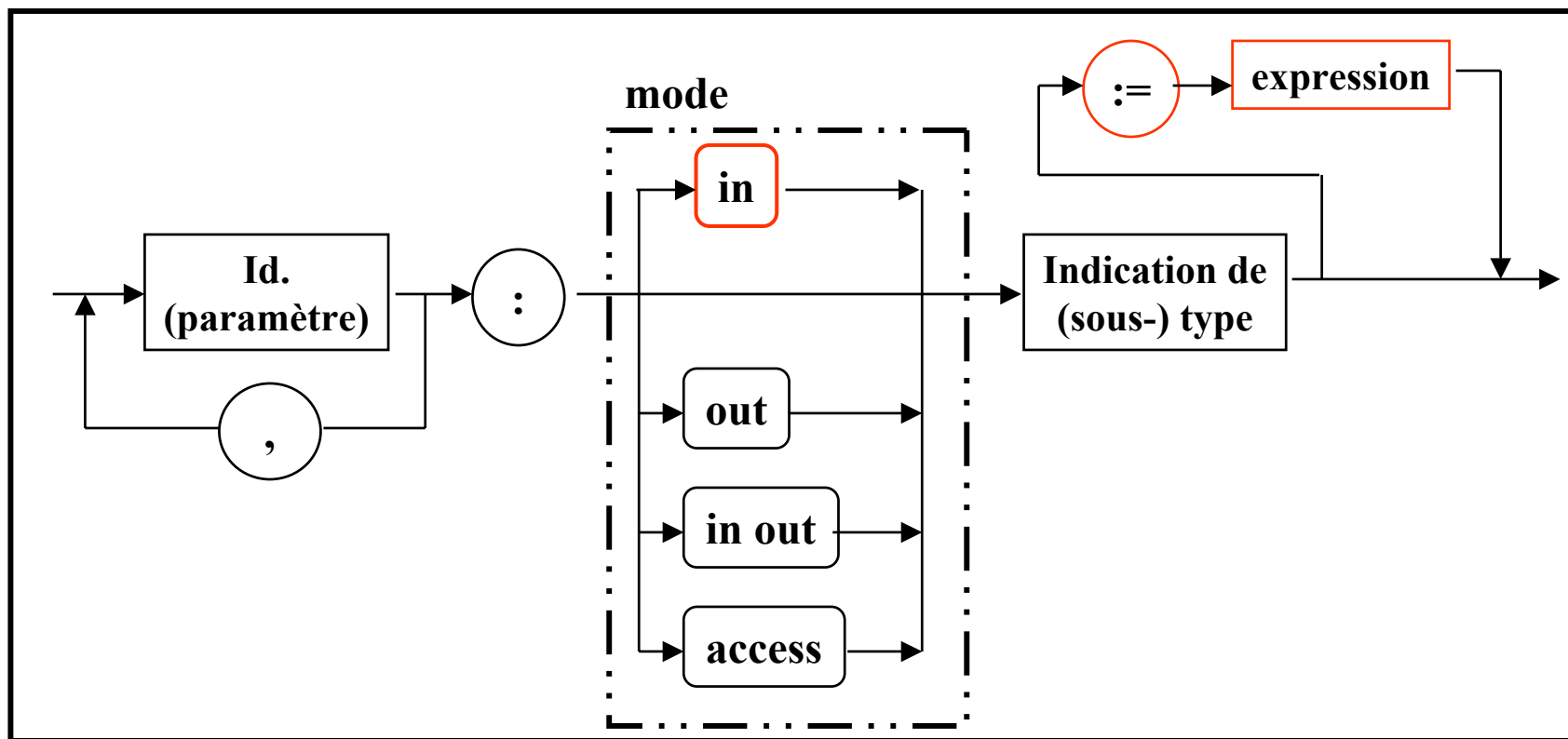


Exemple :

```
procedure Permute(X,Y : in out Integer) is
    Tmp : Integer;           -- garde temporairement X
begin -- algorithme de Permute
    Tmp := X;
    X := Y;
    Y := Tmp;
end Permute;
```


Déclaration de paramètres dans sous-programme

- Une déclaration de sous-programme définit son nom et la manière dont on l'appelle, c'est à dire dont on fait exécuter les ordres qui sont dans son corps.
- La manière dont on appelle un sous-programme dépend essentiellement de la déclaration de ses paramètres éventuels.



Traduction des paramètres formels

Notation algorithmique

ADA

Consulté X : typeparam

Elaboré Y : typeparam

Modifié Z : typeparam

X : **in** typeparam

Y : **out** typeparam

Z : **in out** typeparam

- Le paramètre est consulté : on met **In** devant son type
- Le paramètre est élaboré : on met **Out** devant son type
- Le paramètre est modifié : on met **In Out** devant son type

Remarque : le langage Ada est très proche de l'algorithmique

- Pour chaque paramètre formel :
 - nom
 - mode
 - ✓ **in** : lecture seule
 - ✓ **in out** : lecture / mise à jour
 - ✓ **out** : écriture ou mise à jour seule
 - type

- Par défaut le mode est **in**
- Un paramètre de mode **in** est considéré comme une constante
- Les fonctions n'autorisent que le mode **in**.

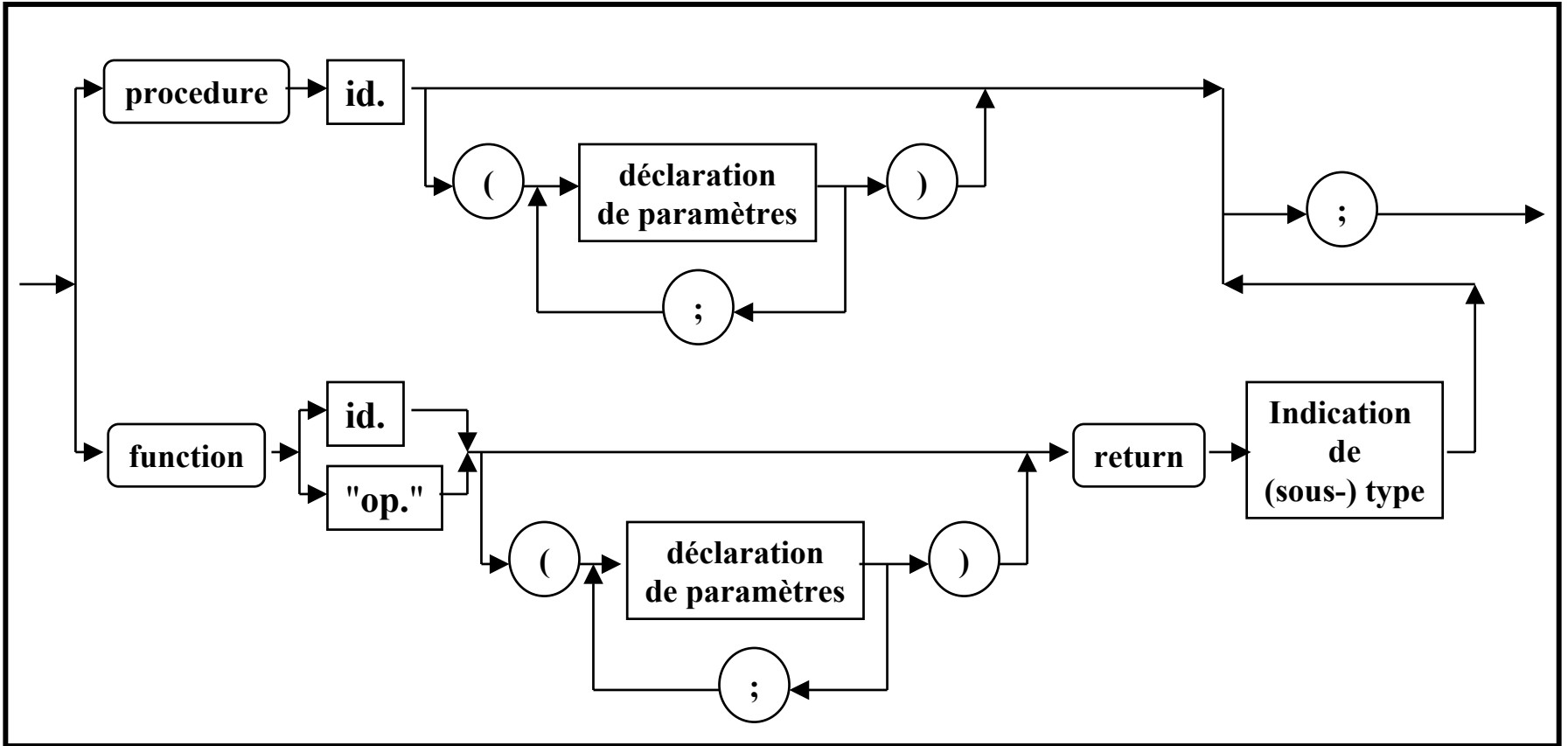
Exemple : on peut déclarer

```
X : in Float := 1.2;
Y : in Float := 1.5;
Z : out Float -- ( valeur de retour)
```

Convention :

On commence par les paramètres de mode **in**, puis **in out** et enfin **out**

- La déclaration de *sous programme* est définie par le diagramme suivant :



Exemple de specifications

- Soit le type : **type** TDecimal **is range** 1 .. 10 ;
- Une *fonction* est un sous programme qui, lorsqu'il est appelé, exécute ses ordres et retourne une valeur se substituant à l'appel. Le (sous-) type du *résultat* est donc indiqué après le mot **RETURN**.

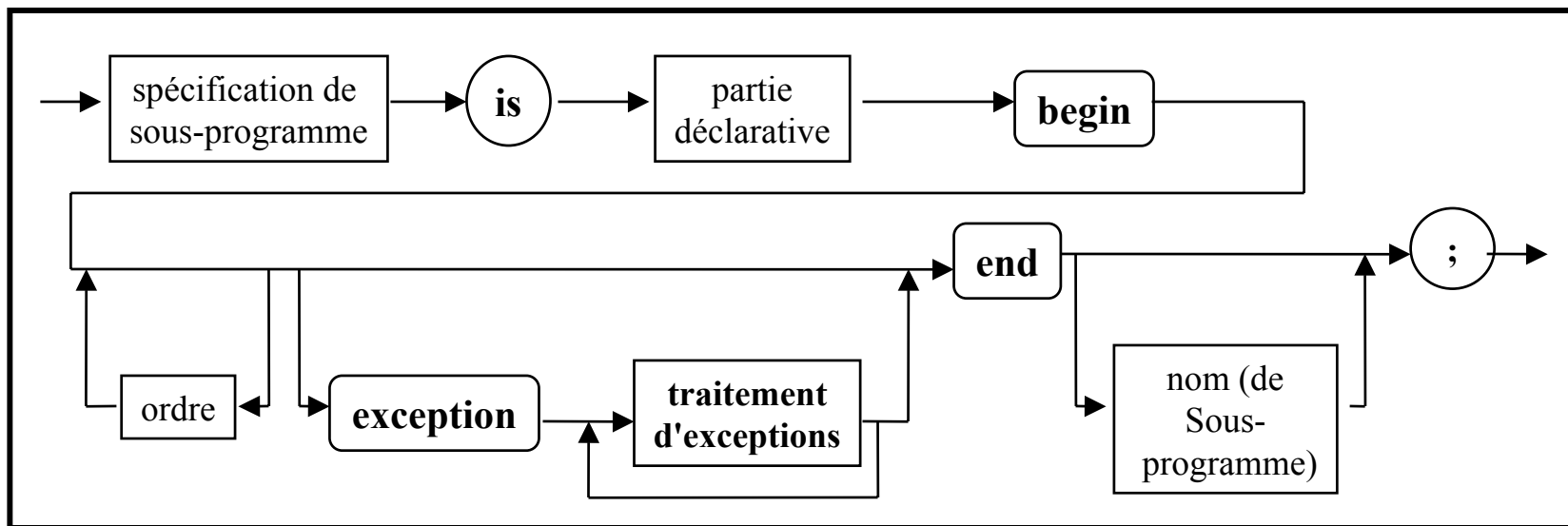
function Somme(X, Y : **in** TDecimal) **return** TDecimal ;

- Une *procedure* est un sous programme qui, lorsqu'il est appelé, exécute ses ordres en communiquant des valeurs par ses paramètres. Elle est donc déclarée par les seules indications de son nom et de ses paramètres.

procedure Somme(X, Y : **in** TDecimal ; Z : **out** TDecimal);

Corps (body) de sous programme

- Le *corps* d'un sous-programme est défini par le diagramme suivant.



- La spécification de *sous-programme* présente au début du corps de sous-programme doit être identique à celle de la déclaration
- Cette déclaration peut être omise, sauf si le sous-programme est utilisé dans un progiciel, sa déclaration devant être dans la partie visible (et son corps dans la partie cachée), ou bien si le corps du sous programme est placé après celui d'une unité qui l'utilise.

- Déclaration

```
function Somme(X, Y : in TDecimal) return TDecimal ;
```

- Corps

```
function Somme(X, Y : in TDecimal) return TDecimal is
```

```
    Z : TDecimal ; -- Partie déclarative
```

```
begin
```

```
    Z:= X+Y ; -- Partie instructions
```

```
    return Z ; -- On peut utiliser return X+Y
```

```
end Somme;
```

Une fonction s'achève avec une instruction **return**.

Une fonction peut contenir plusieurs instructions **return**.

- Déclaration

```
procedure Somme(X, Y : in TDecimal ; Z : out TDecimal);
```

- Corps

```
procedure Somme(X, Y : in TDecimal ; Z : out TDecimal) is
```

```
    -- Partie déclarative ici aucun besoin
```

```
begin
```

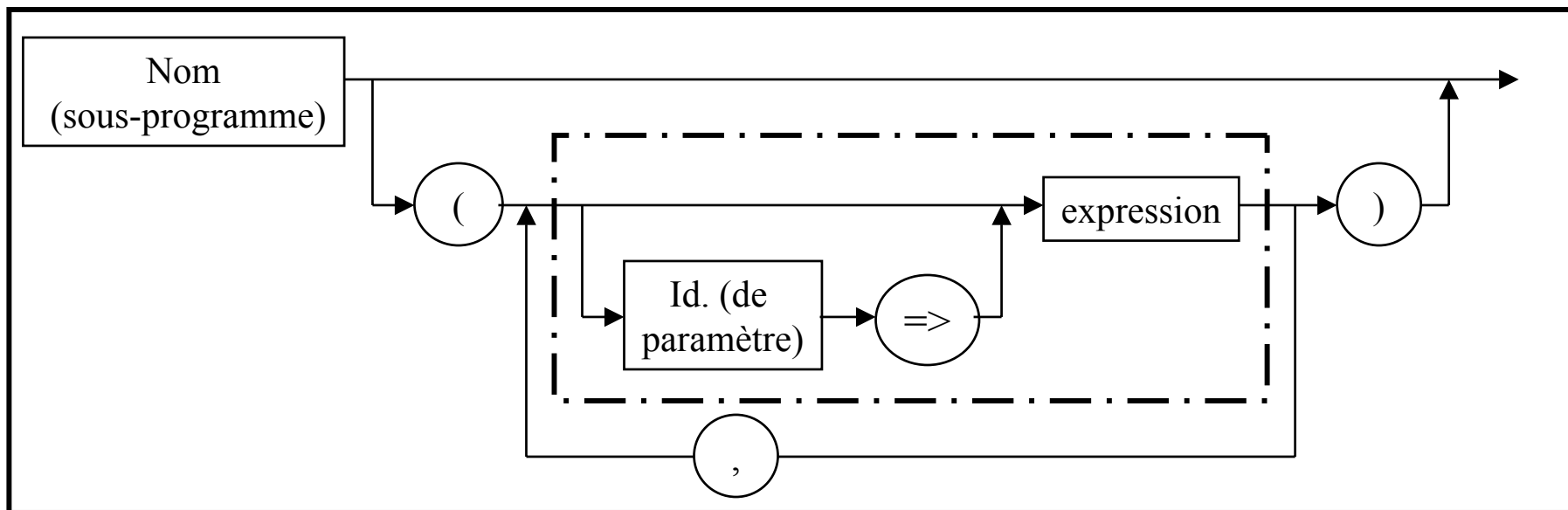
```
    Z := X+Y; -- Partie instructions
```

```
end Somme;
```

Une procédure s'achève avec le **end** final.

Appel de sous programme

- L'appel de *procédure ou fonction*, est un *ordre* simple défini par le diagramme suivant



- Cet ordre entraîne l'exécution de la procédure dont le nom est donné, après avoir donné à chaque paramètre IN ou IN OUT une valeur, dites argument d'*appel*, sauf éventuellement si le paramètre correspondant à une valeur par défaut.

```
Get(A);
Z := Sin(X+Y);
```

```
New_Line(Spacing =>3);
X := Random(Generator =>Generateur);
```

```
Skip_Line;
```

Remarque : La *procedure* **Somme** communique son résultat par le paramètre *Z*, alors que la *fonction* **Somme** retourne elle-même le résultat, et est donc directement utilisable dans une expression.

■ Fonctions

- appelées au sein d'expression
- retourne une valeur

■ Procédures

- appelées comme des instructions,
- modifie une valeur

■ Déclaration et Corps

- déclaration : convention d'appel (optionnelle)
- corps : actions à effectuer (au moins une instruction "**null**")

Arguments d'appel

▪ La liste des arguments d'appel peut être :

- *positionnelle*, c'est à dire que les valeurs des arguments sont données respectivement et dans l'ordre aux paramètres (tel qu'ils sont déclarés dans la spécification de procédure) ;

- *nommée*, les valeurs étant données aux paramètres dont le nom les précède, séparé par " => " L'ordre est alors quelconque ;

- *mixte*, la partie positionnelle précédant la partie nommée.

Exemple : la procédure "Agenda" imprime le calendrier d'un mois, de jour à jour, à partir d'une date donnée est déclarée ainsi :

procedure Agenda (An, Mois : **in** natural; Jour : **in** natural := 1) ;

Agenda(1999, 3, 8) ;

-- *positionnelle*

Agenda(Jour => 8, Mois => 3, An => 1999) ;

-- *nommée*

Agenda(1999, Jour => 8, Mois => 3) ;

-- *mixte*

Agenda(Mois => 3, An => 1999) ; ou Agenda(1999, 3) ;

-- *la valeur du Jour = 1*

Concordance de type

En principe le type du paramètre effectif doit être identique à celui du paramètre formel. Toutefois, dans la mesure du raisonnable, on peut demander une conversion dans un sens au moment de l'appel et dans le sens inverse au retour.

- Imaginons l'en-tête de procédure:

procedure Cube (I : **in out** Integer) **is**

...

- dans le module qui appelle Cube on a une variable J déclarée INTEGER on peut donc appeler simplement :

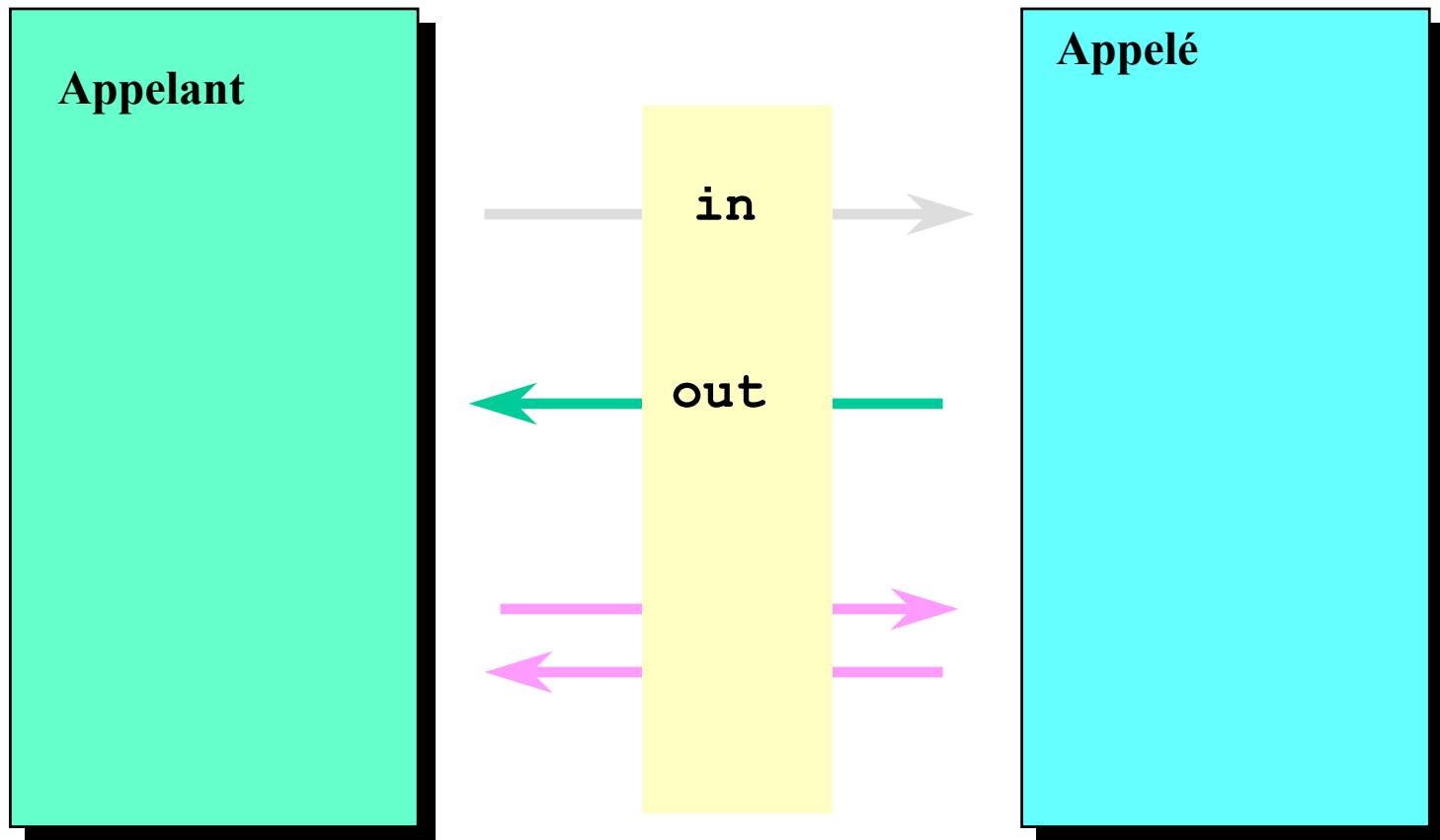
Cube (J);

- Mais si l'on a aussi une variable X de type FLOAT, on peut également appeler:

Cube (Integer(X));

- il y aura alors conversion **FLOAT** $\Leftarrow\Rightarrow$ **INTEGER** à l'entrée du sous-programme et conversion inverse au retour.

Appel de sous programme



Signature \equiv contrat entre l'appelant et l'appelé

Appel de sous programme

- L'exécution des ordres du sous programme commence après appel de celui-ci et affectation des valeurs d'appel aux paramètres IN et IN OUT ; lorsqu'elle se termine les paramètres IN OUT et OUT reçoivent les valeurs de retour, et l'unité appelant poursuit le traitement

procedure Transformation is

```

type T_Vect is array (1..3) of Float ;
type T_Mat is array (1..3;1..3) of Float ;
    Mrot, Mtrans, Mhom : T_Mat := ((1.0,0.0,0.0),
                                (0.0,1.0,0.0),
                                (0.0,0.0,1.0));
    V, Vr : T_Vect := (others => 0.0);
begin
    Vr := "*" (Mrot, V);
    AfficherVect(Vr);
    Vr := Mhom * V;
    AfficherVect(Vr);
end Transformation ;
  
```

```

function "*" (M : in T_Mat ;
              Ve : in T_Vect) return T_Vect ;

function "*" (M : in T_Mat ;
              Ve : in T_Vect) return T_Vect is
    Vs : T_Vect := (others => 0.0);
    begin
      for i in M'range(1) loop
        for j in M'range(2) loop
          Vs(i) := Vs(i)+M(i,j)*Ve(j) ;
        end loop;
      end loop;
      return Vs;
    end "*" ;
  
```

Paramètres formels

Partie déclarative

Paramètres effectifs

Partie non visible

Exemple

procedure Trier **is**

A, B, C : Integer;

procedure Trier (X,Y : **in out** Integer);

procedure Saisir(X : **in out** Integer);

procedure Trier (X,Y : **in out** Integer) **is**

Temp : Integer; *-- garde temporairement X*

begin *-- Trier*

if (X > Y) **then**

Temp := X;

X := Y;

Y := Temp;

end if ;

end Trier;

procedure Saisir(X : **in out** Integer) **is**

begin

Put(" Entrer La valeur : ");

Get(X);

Skip_Line;

end;

begin *-- Trier*

Saisir(A); Saisir(B); Saisir(C);

Trier(A, B); Trier(B, C); Trier(A, B);

end Trier ;



On peut remplacer ce code par une procédure `Permuter`

Portées et visibilité

Si une procédure (ou fonction) en appelle une autre qui elle-même appelle à son tour la première, nous avons un petit problème au niveau des déclarations => Problème de la récursivité croisée

procedure F (...);

procedure G (...) is

begin -- G

F (...);

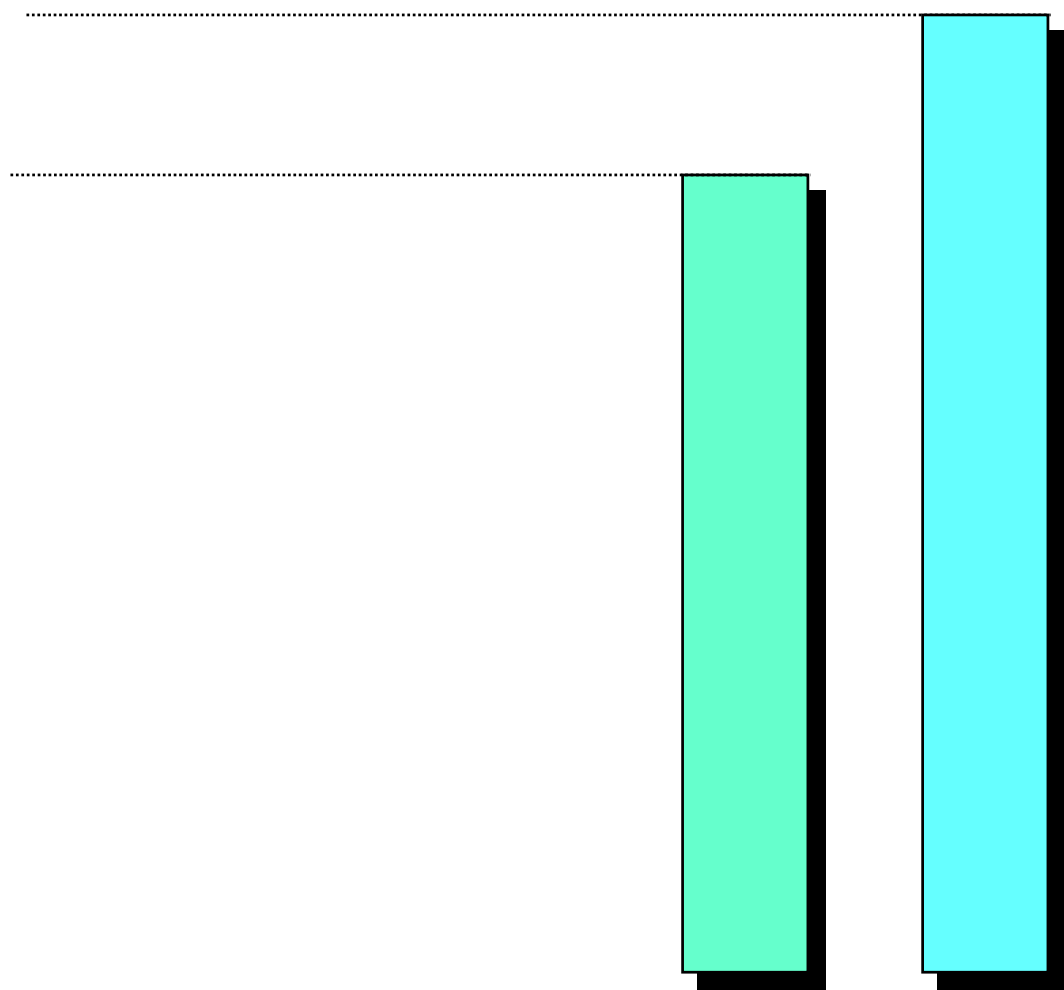
end G;

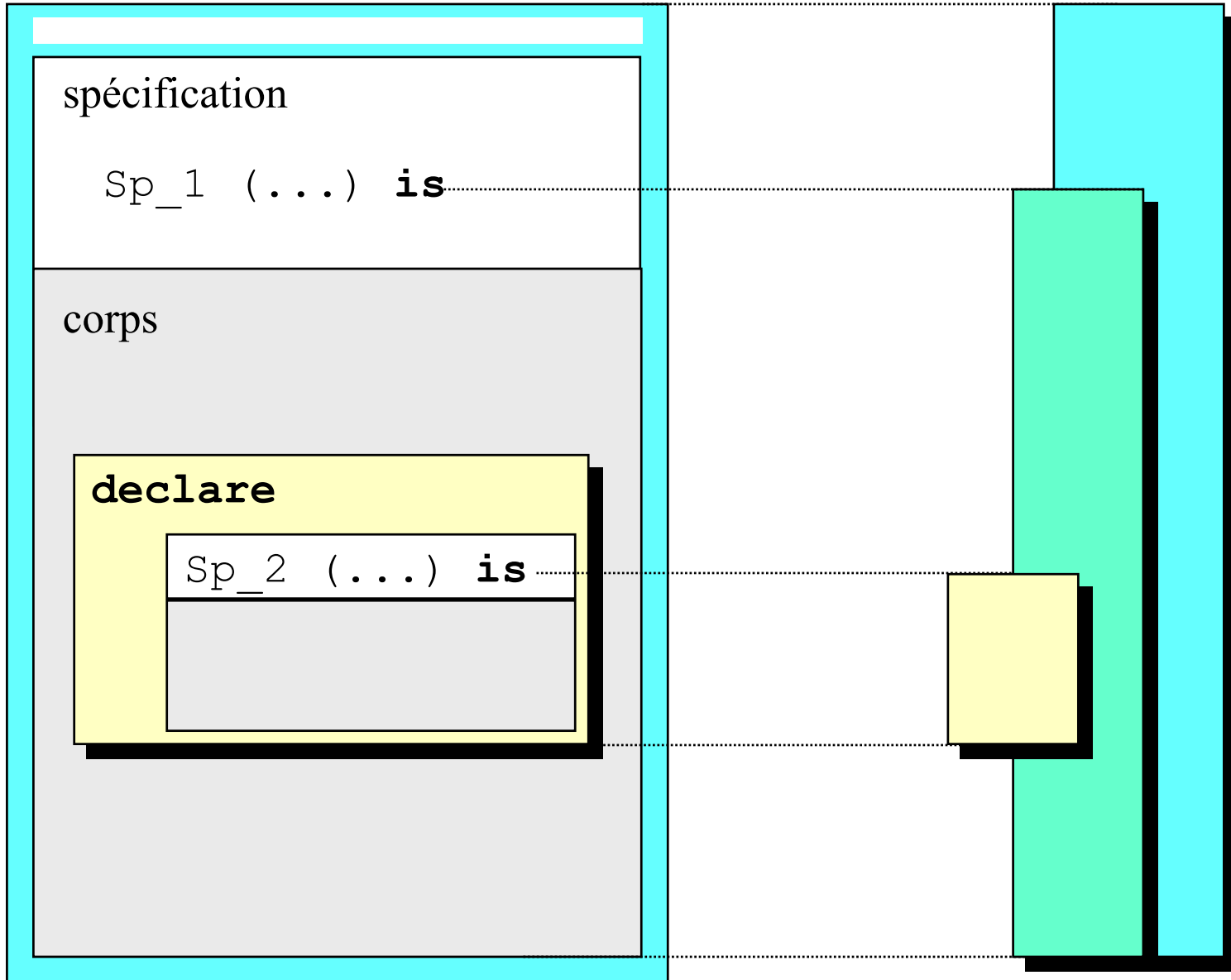
procedure F (...) is

begin -- F

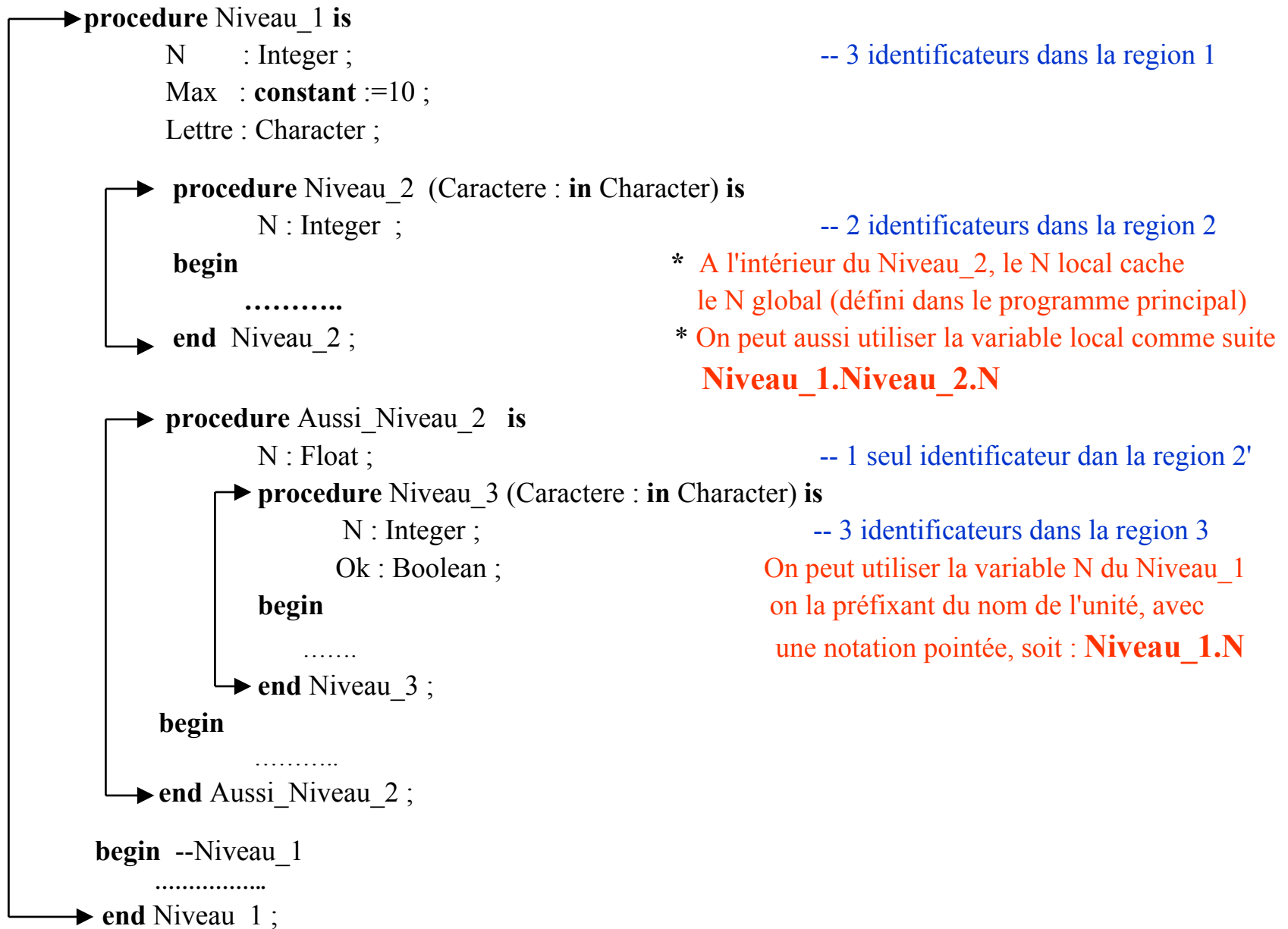
G (...);

end F;





Portée et visibilité des paramètres (Global, Local)



```

procedure Exemple is
  A : Integer := 1;
  function F1 return Integer is
  begin
    return A + 3;
  end F1;

  function F2 return Integer is
  begin
    A := A * 10 ;
    return A + 3;
  end F2;

```

Il ne faut **jamais** utiliser de variables globales, mais les transmettre en paramètres.

```

begin
  Put ( F1 + F2) ; -- 4 + 13 = 17
  A : Integer := 1;
  Put ( F2 + F1) ; -- 13 + 13 = 26
end Exemple;

```

Remarque : $F1 + F2 \triangleleft F2 + F1$

Les effets de bords

- Il en va de même si les fonctions avaient des paramètres de sortie:

procedure Exemple is

 A : Integer := 1;

function F1 (A : **in** Integer) **return** Integer **is**

begin

return A + 3;

end F1;

function F2 (A : "**in out**" Integer) **return** Integer **is**

begin

 A := A * 10 ;

return A + 3;

end F2;

begin

 Put (F1 (A) + F2 (A)) ; -- 4 + 13 = 17

 A : Integer := 1;

 Put (F2 (A) + F1 (A)) ; -- 13 + 13 = 26

end Exemple;

Remarque : $F1 + F2 \not\triangleq F2 + F1$ de même $F2 + F2 \not\triangleq 2 * F2$

Surcharge de sous programmes ou d'opérateurs (polymorphisme statique)

- Le nom d'une fonction est soit un identificateur ordinaire, soit un opérateur prédéfini auquel on attribue un nouveau sens, et que l'on appelle surchargé. Il doit alors figurer en double apostrophe comme une chaîne de caractères littérale.
- Peuvent être surchargés les opérateurs
 - unaires : abs, +, -, not (logique)
 - binaires : **, *, /, +, -, mod, rem, & relation (>, <, <=, >=, =, /=), logique(and, or, xor)

Exemple : `type V is array (1..10) of Float ;` (voir cours tableau)
`function "+"(X, Y : in V) return V ;`

Le corps de la fonction définira l'addition de 2 vecteurs, et l'on pourra alors additionner deux objets A et B de type V par A + B.

Remarque :

Un opérateur surchargé ne peut avoir de paramètre avec valeur initiale par défaut.

Surcharge d'opérateurs (polymorphisme statique)

Définition de 2 sous-programmes ayant le même nom mais des signatures différentes.

```
function "+" (Left, Right : Integer) return Integer;
```

```
function "-" (Left, Right : Integer) return Integer;
```

```
function "+" (Left, Right : Float) return Float;
```

```
function "-" (Left, Right : Float) return Float;
```

```
function "+" (Left, Right : T_Complexe) return T_Complexe;
```

```
function "-" (Left, Right : T_Complexe) return T_Complexe;
```

Identification : à partir des paramètres d'appels

Sommaire

- Chapitre I : Présentation
- Chapitre II : Unités lexicales
- Chapitre III : Types et sous types
- Chapitre IV : Ordres (Sélection, cas, itération, ...)
- Chapitre V : Sous programmes
- **Chapitre VI : Tableaux (array)**
- Chapitre VII : Chaînes de caractères (String)
- Chapitre VIII : Articles (record)
- Chapitre IX : Pointeur
- Chapitre X : Fichiers (File)
- Chapitre XI : Paquetages simples (package)
- Chapitre XII : Généricité
- Chapitre XIII : Tâches

Chapitre VI : Tableaux

- Introduction au type tableau
- Attributs First, Last, Length et Range
- Présentation d'un type tableau
- Initialisation
- Accès à un élément particulier du tableau
- Présentation de types spécifiques
- Agrégats
- Opérations sur les tableaux

Introduction au type tableau.

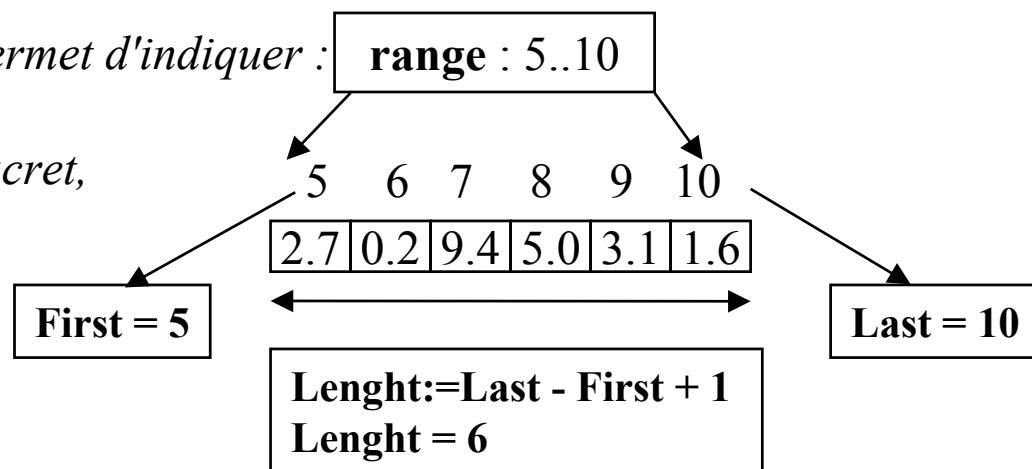
Attributs First, Last, Lenght et Range

- *Un tableau est un ensemble de composantes de même type dont chacune est repérée par un ou plusieurs indices.*

La spécification d'un type tableau permet d'indiquer :

- le nombre d'indices N ,
- le type des indices qui doit être *Discret*,
- le type des éléments.

dimension $N:=1$
 indice entier de 5 à 10
 contenant 6 réels



- Les attributs *First*, *Last*, *Lenght*, et *Range* sont applicables aux indices d'un tableau T

$T'First(N)$ donne la première valeur d'indice de la dimension N du tableau T

$T'Last(N)$ donne la dernière valeur d'indice de la dimension N du tableau T

$T'Lenght(N)$ donne la longueur de la dimension N du tableau T

$T'Range(N)$ représente l'intervalle de la dimension N du tableau T

- Les types composés : les tableaux

```
type T_Fruits is (Orange, Banane, Mandarine, Kiwi);
```

Objet tableau

```
Mon_Tableau : array (1 .. 5) of T_Fruits;
```

```
-- L'objet Mon_Tableau est de type anonyme
```

Type tableau

```
type T_Tab_Fruits is array (1 .. 5) of T_Fruits;
```

```
Mon_Tableau : T_Tab_Fruits ;
```

```
Un_Tableau   : T_Tab_Fruits ;
```

Initialisation

```
Mon_Tableau := (1|3|5 => Orange, others => Banane);
```

```
Mon_Tableau := (1..3 => Orange, others => Banane);
```


Présentation d'un type tableau

- Le type d'indice doit être discret alors que le type des éléments peut être n'importe lequel sauf un type (ou sous type), un tableau non contraint ou un article à discriminants sans valeurs par défaut
- **array (Integer range $\langle \rangle$) of Float** \Leftrightarrow **array (Integer) of Float**
 - matrice d'entiers, bornes des indices indéfinies
- **array (Integer range $\langle \rangle$, Integer range $\langle \rangle$) of Integer**
- **array (Natural range 1..100) of Float** *vecteur de cent composantes réelles*
- **array (1..100) of Float** *idem*
 - Idem, mais contrainte sur les composantes
- **array (1..100) of Float delta 0.1 range -10.0..10.0**
- **array (Boolean) of Character** (*vecteur à indice, False ou True*) *de caractères*
- **array (Boolean, Integer range 1..3) of Float** *matrice avec six composantes réelles*
- **array (Natural range $\langle \rangle$) of Character range 'A'..'Z'** *un tableau caractères*

Notation algorithmique

T : tableau sur [bi..bs] de typeéléments

Exemples :

T : tableau sur [1..10] d'entiers = [0,-1, 0, 1, 2, 3, 4, 4, 4, 4]

**P : tableau sur [1..2,1..3] d'entiers = [[1,2,3],
[4,5,6]]**

ADA

T : array(bi..bs) of typeelements;

Exemples :

T : array(1..10) of Integer := (0,-1,0,1,2,3,others => 4);

P : array (1..2,1..3) of Integer := ((1,2,3),(4,5,6));

Notation algorithmique

Désignation d'un élément de tableau :

$T[i]$ désigne l'élément d'indice i du tableau T .

ADA

$T(i)$ désigne l'élément d'indice i du tableau T .

Remarque :

T : array(1..10) of Integer;

begin

**$T(12) := 3;$ -- *génère une erreur, dépacement d'indice*
 -- *un code plus sûr***

Tableau non contraint

Présentation de types spécifiques

Pour déclarer un nouveau *type spécifique* en utilisant *le type* de base *array*

Un seul type non contraint, les variables déclarées sont obligatoirement contraintes

```
type T_Vecteur is array(Integer range <>) of Float ; -- tableau non contraint  
-- <> se dit "boite"
```

Il est possible de déclarer un *sous-type* d'un type (de base) tableau non contraint. La forme générale d'une telle déclaration est :

```
subtype identificateur is id_type_tableau (Intervalle_1.. Intervalle_N ) ;
```

- identificateur est le nom du sous-type
- id_type_tableau est le nom d'un type tableau non contraint
- intervalle_i est un intervalle fixant les bornes

```
subtype Vecteur_5D is T_Vecteur(1..5); -- tableau de réels à 5 composantes.
```

Exemple : somme de 2 vecteurs

```

with ada.Text_IO; with ada.Integer_Text_IO; use ada.Text_IO; use ada.Integer_Text_IO;
procedure VecteurSomme is
    N : Integer ;                                -- dimension du tableau
    type Vecteur is array (Integer range<>) of Float; -- nouveau type vecteur
begin --VecteurSomme
    Put(" Entrer la dimension des vecteurs : "); -- Saisie de la dimension des vecteurs
    Get(N);
    Skip_Line;
    Somme : declare
        Vect_Somme, Vecteur_1, Vecteur_2 : Vecteur(1..N); -- variables de type vecteur
    begin
        Put_Line(" Saisie des composantes de vecteur_1 et vecteur_2"); -- Saisie des vecteurs
        for I in 1..N loop
            Get(Vecteur_1(I)); Get(Vecteur_2(I));
            Skip_line;
        end loop;
        for I in Vect_Somme'First .. Vect_Somme'Last loop -- Vect_Somme'Range
            Vect_Somme(I):=Vecteur_1(I)+ Vecteur_2(I);
        end loop;
    end Somme ;
end VecteurSomme;

```


Agrégats

- Un *agrégat* est la forme littérale d'une valeur de type tableau (un élément de l'ensemble de valeurs du type tableau). Il existe deux formes d'agrégats qui ne doivent pas être mélangées :

- **l'agrégat positionnel** où les valeurs sont données dans l'ordre où elles seront stockées dans la variable de type tableau. Les valeurs sont séparées par des virgules.

`v : Vecteur_5D := (1.0, 2.0, 3.0, 4.0, 5.0);`

- **l'agrégat par nom** où chaque valeur est précédée de la valeur de l'indice et du symbole `=>`, ainsi l'ordre des indices n'a plus à être respecté.

`v : Vecteur_5D := (1 => 1.0, 2 => 2.0, 4 => 4.0, 5 => 5.0, 3 => 3.0);`

Remarque : Pas d'agrégat mixte, seule la clause "others" est autorisée

Les règles de formation des agrégats ressemblent à celles utilisées pour l'instruction `case`.

Chaque valeur peut être donnée pour un seul indice ou un intervalle ou plusieurs indices donnés explicitement.

La clause others permet de donner une valeur à tous les éléments non spécifiés mais doit toujours être placée à la fin de l'agrégat :

`w : Vecteur(1..1000) := Vecteur'(1 => 1.0, 2 | 4 => 2.0, 3 => 3.0, 7..15 => 5.0, others => 0.5);`

`m3 : Mat_3_3 := (1=> (1=>1.0, others=>0.0), 2=> (2=>1.0, others=>0.0), 3=> (3=>1.0, others=>0.0));`

`v : Vecteur(1 .. 15) := (1.0, 2.0, 3.0, 4.0, 5.0, others => 0.0);`

La déclaration multiple suivie d'une affectation provoque l'initialisation des deux tableaux :

`v1, v2 : Vecteur_5D := (others => 1.0);` initialise toutes les valeurs de `v1` et de `v2` à 1.0.

Opérations sur les tableaux

- Affectation
- Tranches de tableaux
- Concaténation
- Comparaison (égalité)
- Comparaison
- Opérations booléennes
- Conversion

Affectation

- Pour pouvoir affecter un tableau à un autre il est nécessaire qu'ils soient de même type, et de même taille pour chaque dimension, sans avoir forcément les mêmes indices, faute de quoi l'exception **Constraint_Error** sera levée .
- Quand les premiers indices ne sont pas identiques, on parle de glissement

Exemples :

T : **array**(1..5) **of** Integer := (0,-1,0,1,2);

P : **array**(6..10) **of** Integer;

L : **array**(6..11) **of** Integer;

P := T; est correct;

L := T; incorrect; --levée d'une exception, **Constraint_Error**

type Vecteur **is** **array**(Integer **range** <>) **of** Integer;

V : Vecteur (1..5) := (0,-1,0,1,2);

W : Vecteur (0..4);

W := V;

Tranches de tableaux

Une *tranche* (slice) de tableau à une dimension est un *tableau partiel* défini par un intervalle compris dans l'intervalle de définition des indices du tableau d'origine.

Exemple :

```
T : array(1..8) of Integer := (2,-1,0,1,2,5,3,2);
P := T(3..5);  -- contenu de P est de (0,1,2)
P'First = 3;
P'Last = 5;
```

Remarque :

Il ne faut pas confondre $P(0)$ et $P(0..0)$

$P(0)$: est l'élément de la tranche P à la position d'indice 0

$P(0..0)$: est une tranche, donc un tableau à un seul élément dont l'indice est 0

Concaténation

- L'opération de *concaténation* consiste à mettre bout à bout des tableaux ou tranches de tableaux de même type. Cet opérateur est noté `&`, et possède la priorité des opérateurs binaire (`+` et `-`).
- Le tableau résultat doit avoir autant d'éléments que la somme des élément des deux tableaux à concaténer.

Exemple :

```
type Vecteur is array( Integer) of Float; -- type non contraint
```

```
W : Vecteur (-10..10) := (others => 0.0);
```

```
X : Vecteur (1..10) := (1.5, 2.5, 3.5, others => 4.0);
```

```
V1 : Vecteur(1..10) := W (1..5) & X (1..5); -- utilise les tranches de tableaux
```

```
V2 : Vecteur := W & X; -- V2 aura la borne inférieure de W et 31 éléments
```

Comparaison (Égalité)

- *Égalité, inégalité* $= \neq$: il est possible de comparer des tableaux de même type et de même longueur pour chaque dimension, sans avoir forcément les mêmes indices.
- Les tests d'égalité suivent les mêmes règles de glissement que l'affectation sauf que si les tableaux n'ont pas la même longueur, la valeur retournée sera *false*.
- Les deux tableaux seront égaux s'ils possèdent le même nombre d'éléments et que ceux-ci soient égaux (en respectant l'ordre des éléments).

Exemple :

$(1, 2) = (1, 3, 5, 7)$	expression fausse; -- <i>pas la même longueur</i>
$(2, 1) \neq (1, 3, 5, 7)$	expression fausse; -- <i>pas la même longueur</i>
$(2, 1) = (2, 1)$	expression vraie;
$(2, 0) = (2, 1)$	expression fausse;

Remarque :

U,V is array(1..5) of Float := (others => 0.0); -- *type anonyme*
if U = V **then**(provoque une erreur car U et V sont de type différent)

Comparaison

- Opérateurs de *comparaison* $<$, $<=$, $>$, $>=$: s'appliquent uniquement à des tableaux à une dimension et contenant des éléments de type *discret*.
- Durant l'opération de *comparaison*, les éléments sont comparés un à un jusqu'à épuisement de l'un des tableaux ou jusqu'à ce que l'une des comparaisons permette de répondre à la question.

Exemple :

-- *expression qualifiée*

String('a', 'b') < ('a', 'b', 'c', 'd') expression vraie; -- *longueur 1 < longueur 2*

String('a', 'b') > ('a', 'b', 'c', 'd') expression fausse; -- *expression qualifiée*

(1, 2) < (1, 3, 5, 7)	expression vraie; -- <i>1 < 2 et 2 < 3 + L1 < L2</i>
(2, 1) < (1, 3, 5, 7)	expression fausse; -- <i>2 > 1</i>
(2, 1) > (1, 3, 5, 7)	expression vraie; -- <i>2 > 1</i>
(2, 1) > (2, 1)	expression fausse;
(2, 1) >= (2, 1)	expression vraie; -- <i>2 = 2</i>
(2, 0) >= (2, 1)	expression vraie; -- <i>2 = 2</i>

Opérations booléennes

Les opérateurs logiques *and*, *or*, *xor* et *not* s'appliquent aux tableaux unidimensionnels de même longueur dont le type des éléments est le type *Boolean*.

L'application d'un opérateurs sur un ou des tableaux se fait élément par élément de telle manière à générer un tableau résultat de même longueur que le ou les tableaux initiaux.

Exemple :

not (False , True, True)	résultat (True, False, False)
(True , True) and (False, True)	résultat (False, True)
(True , True) or (False, True)	résultat (True, True)
(True , True) xor (False, True)	résultat (True, False)

Remarque : Les bornes du tableau résultat sont celles de l'unique opérande ou de l'opérande de gauche

Conversion entre types tableaux

- Il faut que :
- les types aient le même nombre de dimensions;
 - les types d'indices sont identiques ou convertibles entre eux;
 - les types des éléments sont identiques.

type T_Vecteur **is array** (Integer **range** $\langle \rangle$) **of** Float;

subtype T_Vecteur_10 **is** T_Vecteur (1..10);

type T_Autre_Vecteur **is array** (Natural **range** $\langle \rangle$) **of** Float;

Vecteur : T_Autre_Vecteur (0..100);

Vecteur_Test : T_Vecteur (-100..100);

T_Vecteur (Vecteur)	bornes de Vecteur, 0 et 100;
T_Vecteur (Vecteur(20..30))	bornes de la tranche, 20 et 30;
T_Vecteur_10 (Vecteur(20..29))	bornes de T_Vecteur_10, 1 et 10;

T_Autre_Vecteur (Vecteur_Test)
 provoque Constraint_Error à l'exécution (-100..0 hors de Natural);

T_Vecteur_10 (Vecteur(20..30))
 provoque Constraint_Error à l'exécution (longueurs différentes).

Sommaire

- Chapitre I : Présentation
- Chapitre II : Unités lexicales
- Chapitre III : Types et sous types
- Chapitre IV : Ordres (Sélection, cas, itération, ...)
- Chapitre V : Sous programmes
- Chapitre VI : Tableaux (array)
- **Chapitre VII : Chaînes de caractères (String)**
- Chapitre VIII : Articles (record)
- Chapitre IX : Pointeur
- Chapitre X : Fichiers (File)
- Chapitre XI : Paquetages simples (package)
- Chapitre XII : Généricité
- Chapitre XIII : Tâches

Chapitre VII : String

- VII.1 Présentation du type prédéfinie String
- VII.2 Attributs Image et Value
- VII.3 Exemple de chaînes de caractères
- VII.4 Manipulation de chaînes de caractères

VII.1 Présentation du type prédéfinie String

procedure chaine_caractere **is**

type String **is array** (Positive range <>) **of** Character ; -- défini dans le paquetage
subtype Mon_String **is** String(1..20);

Mon_Bonjour : **constant** String := "Bonjour" ;

Au_Revoir : **constant** String := ('a', 'u', ' ', 'r', 'e', 'v', 'o', 'i', 'r') ; -- agreg. Pos.

Chaine_Vide : **constant** String := "";

Taille_Message : **constant** := 20 ;

Message1 : String(1..Taille_Message) ;

Message2 : Mon_String ;

begin

Message1(9) := 'e'; -- Acces aux éléments de la chaine

Message1(Mon_Bonjour'Range) := Mon_Bonjour; --tranche de tableau

Message2(1.. Mon_Bonjour 'Length + Au_Revoir'Length) :=

Message1(Mon_Bonjour'Range) & Au_Revoir ; -- concaténation

end chaine_caractere ;

VII.2 Attributs Image et Value

- **Attributs Image et Value** (valeur scalaire => image de caractères)

Integer'Image(12)	donne la chaîne " 12"
Integer'Image(-12)	donne la chaîne "-12"
Float'Image(12.34)	donne la chaîne " 1.23400E+01"
Character'Image('a')	donne la chaîne "'a'"
T_Mois_D_L_Année'Image(Mars)	donne la chaîne "MARS"
Integer'Value(" 12")	donne le nombre entier 12
Integer'Value("-12")	donne le nombre entier -12
Float'Value(" 1.234E+01")	donne le nombre réel 1.234E+01
Character'Value("'a'")	donne le caractère 'a'
Natural'Value("-12")	lèvera la contrainte <i>Constraint_Error</i>

- **Entrées–sorties telles qu'elles sont définies dans Ada.Text_IO**

```

procedure Put( Item : in String);
procedure Put_Line(Item : in String);
procedure Get(Item : out String);
procedure Get_Line(Item : out String;
                    Last : out Natural);

```

VII.3 Exemple de chaines de caractères

```
with Ada.Text_IO; use Ada.Text_IO;
```

```
procedure Exemple
```

```
  Bienvenue : constant String := "Bienvenue dans l'exemple!";
```

```
  Taille    : constant := 10;
```

```
  Chaine    : String(1..Taille); -- Chaine de 10 caractères au maximum
```

```
  Nombre_Car_Lus : Natural ;
```

```
begin -- Exemple
```

```
  Put_Line(Bienvenue);
```

```
  Put("Lecture d'une chaine avec Get(tapez"&  
      Integer'Image(Taille) & "caractères aux moins):"); --1
```

```
  Get(Chaine); --2
```

```
  Skip_Line; --3
```

```
  Put_Line("Voici la chaine que vous avez tapee:" & Chaine);
```

```
  Put_Line("Lecture d'une chaine avec Get_Line(terminez par une");
```

```
  Put(" fin de ligne avant" & Integer'Image(Taille)& "caracteres):");
```

```
  Get_Line(Chaine, Nombre_Car_Lus); --4
```

```
  Put( "Voici la chaine que vous avez tapee: "); --5
```

```
  Put_Line(Chaine(1..Nombre_Car_Lus)); ...
```

```
end Exemple;
```

VII.4 Manipulation de chaînes de caractères

Ada.Strings.Fixed (paquetage sur les traitements de chaînes de caractères)
(opérations disponibles dans ce paquetage)

- **Index**, fonction qui recherche un motif dans une chaîne et qui retourne la position du premier caractère du motif dans la chaîne
- **Count**, fonction qui compte le nombre d'occurrences d'un motif dans la chaîne
- **Insert**, fonction ou procédure qui insert un motif dans une chaîne à partir d'une certaine position
- **Overwrite**, fonction ou procédure qui substitue une partie d'une chaîne par un motif à partir d'une certaine position
- **Delete**, fonction ou procédure qui supprime une partie d'une chaîne

VII.4 Manipulation de chaînes de caractères

```

with Ada.Text_IO;           use Ada.Text_IO;
with Ada.Integer_Text_IO;   use Ada.Integer_Text_IO;
with Ada.Strings.Fixed;     use Ada.Strings.Fixed;

...

procedure Exemple
  Texte : String := "Texte extra dans ce contexte";
  Ext   : constant String := "ext";  -- 2 motifs
  Tex   : constant String := "tex";

begin -- Exemple
  Put(Index(Texte, Ext)) ;           -- Affiche 2
  Put(Index(Texte, Tex)) ;           -- Affiche 24
  Put(Index(Texte, "Tex")) ;         -- Affiche 1
  Put(Count(Texte, "ext")) ;         -- Affiche 3
  Insert(Texte,12,"bleu ciel");      --Texte vaut maintenant "Texte extrableu ciel dans ce"
  Overwrite (Texte,12,"-ordinaire"); --Texte vaut maintenant "Texte extra-ordinaire dans ce"
  Delete(Texte,12,Texte'Last);       --Texte vaut maintenant "Texte extra          "
end Exemple;

```

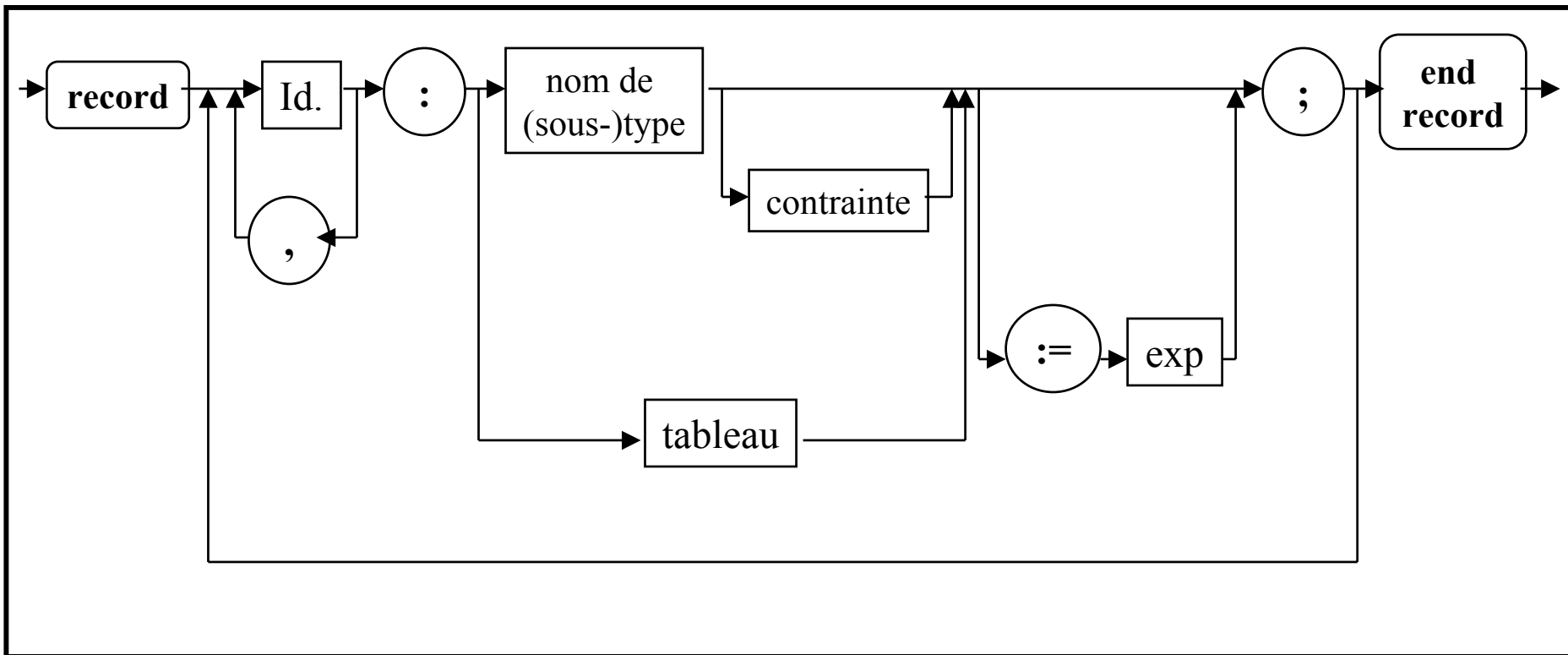

Sommaire

- Chapitre I : Présentation
- Chapitre II : Unités lexicales
- Chapitre III : Types et sous types
- Chapitre IV : Ordres (Sélection, cas, itération, ...)
- Chapitre V : Sous programmes
- Chapitre VI : Tableaux (array)
- Chapitre VII : Chaînes de caractères (String)
- **Chapitre VIII : Articles (record)**
- Chapitre IX : Pointeur
- Chapitre X : Fichiers (File)
- Chapitre XI : Paquetages simples (package)
- Chapitre XII : Généricité
- Chapitre XIII : Tâches

- Présentation d'un enregistrement (Article)

- Présentation d'un type enregistrement sans discriminants
 - Traduction des structures
 - Traduction des types nommés
 - Agrégats et opérations sur les articles
 - Affectation et passage en paramètre de valeurs d'un type article

- Un *enregistrement* ou *un article* est un ensemble de composantes de (sous-) type quelconques appelés *champ*, dont chacune est repérée par son nom.



■ Les types composés : les articles

Objet article : l'ensemble du numéro de rue, du nom de rue, du code postal et du nom de la ville est un objet article.

record

```

Le_Numero      : T_Numeros_Rues;
La_Rue         : T_Noms_Rue;
Le_Code       : T_Codes_Postaux;
La_Ville      : T_Noms_Villes;
    
```

end record

Objet article : l'ensemble de la partie réelle et de la partie imaginaire est aussi un objet article.

record

```

Partie_Reelle   : Float digits 5;
Partie_Imaginaire : Float digits 5;
    
```

end record

Types record de types distincts

type T_Adresses is

record

Le_Numero : T_Numeros_Rues;

La_Rue : T_Noms_Rue;

Le_Code : T_Codes_Postaux;

La_Ville : T_Noms_Villes;

end record;

type T_Point is

record

Abscisse : Float **digits 5 := 0.0;**

Ordonnee : Float **digits 5 := 0.0;**

end record;

Initialisation et utilisation du type article

declare

-- agrégat par position

```
Mon_Adresse : T_Adresses := (10,Marcel,1400,Yverdon);
```

-- agrégat par nom

```
Mon_Adresse : T_Adresses := ( Le_Numero      => 10,  
                             La_rue         => Marcel,  
                             Le_code        => 1400,  
                             La_Ville      => Yverdon);
```

begin

-- Initialisation par agrégat ou champ par champ

```
Mon_Adresse.Le_Numero := 10;
```

end;

Notation algorithmique

Définition d'une structure dans le lexique :

T_M : < A : entier entre 1 et 99 , B : réel >

ADA

```

type T_M is                                -- T_M est de type enregistrement
  record
    A : Integer range 1..99;
    B : Float;
  end record ;
  
```

Notation algorithmique

Désignation d'un champ :

$T_M.A$ désigne le champ A de la structure T_M

ADA

$T_M.A$ désigne le champ A de la structure T_M

Même notation en ADA et en Notation algorithmique

Notation algorithmique

point : type < x, y : réels >

R : un point avec
R.x = 1 et R.y = 12.2

{ constantes }

S : le point < 12,3 >

{Tableau de points}

TP : le tableau sur [1..3] de points
 [<1,12.2>, <12,3>, <-3,1>]

ADA

type T_Point is

record

x,y : Float;

end record;

R : T_Point ;

R.x := 1.0; R.y := 12.2;

S : **constant** T_Point := (12.0, 3.0);

TP : **constant array**(1..3) of T_Point :=
 ((1.0,12.2),(12.0,3.0),(-3.0,1.0));

Notation algorithmique

```

T_Jour : type entier sur 1..31
T_Mois : type mois de l'année

T_Date : type < Jour : T_Jour,
           Mois : T_Mois,
           Année : nombre >

D : T_Date { définition }
    
```

ADA

```

type T_Jour is new Integer range 1..31;
type T_Mois is (Jan, Fev, Mar, Avr, Mai,
                 Jun, Jui, Aou, Sep, Oct, Nov, Dec);

type T_Date is
record
    Jour : T_Jour;
    Mois : T_Mois;
    Année : Natural range 1901..2040;
end record;

D : T_Date; -- définition
    
```

```

D : T_Date:=(15, fev,1995)                -- agrégat par position
D : T_Date:=(Mois=>Fev, Jour=>15, Annee =>1995) -- agrégat par nom
D : T_Date:=(15, Fev, Annee=>1995)        -- agrégat mixte
D : T_Date:=(15, Annee=>1995, Mois=>Fev)   -- agrégat mixte

R1 : T_Point:=(0.0, 1.5);
R1 : T_Point:=(0.0, y =>1.5);
R1 : T_Point:=(y =>1.5, x=>0.0);
R1 : T_Point:=( others =>0.0);
R1 : T_Point:=( 1.5*a, y=>a/2.5); -- on suppose que a est un réel
R2 : T_Point;
R2 := (0.0, 1.5);           -- agrégat ou
R2 := T_Point'(0.0, 1.5); -- agrégat qualifié pour éviter une erreur de compilation

```

- Les opérations possibles:

- Affectation
- Passage en paramètre
- = /=

- Remarques générales:

- Les champs d'un type article peuvent être de n'importe quel type.
- Pas d'entrées-sorties prédéfinies sur les articles.
- Tout agrégat doit être complet

- Cas limite

- Un article peut être vide, mais il faut le dire explicitement

```

type Bidon is
record
    null;
end record;

```

Exemple

procedure Volume **is**

type T_Long **is record**

Dx, Dy : Float;

end record;

S1 := T_Long ;

S2 := T_Long := (1.0, 0.5);

Haut : **constant** Float := 2.5;

V1, V2, V3 := Float;

function Surface **is** (S **in** T_Long) **return** Float **is** -- *S : paramètre formel*

begin

return S.Dx * S.Dy;

end Surface;

begin

-- *Qualification souhaitable mais pas indispensable*

S1 := T_Long ' (**others** => 1.0); -- *S1 et S2 paramètres effectifs*

V1 := Haut * Surface(S1); -- *base d'un carré unitaire*

V2 := Haut * Surface(S2)/Float(2); -- *base d'un triangle*

-- *Qualification possible mais pas indispensable*

V3 := Haut * Surface(T_Long ' (0.5,0.5)); -- *base d'un carré*

end Volume;

Présentation d'un (sous-) type enregistrement à discriminants

- Un *discriminants* est un champ délimité par une paire de parenthèses et localisé entre l'identificateur et le mot réservé **is**.
- Le type d'un discriminant doit être de type discret ou accès
- On peut avoir plus d'un discriminant séparés par ";"

```
type T_Intervalle is range 0..100;
```

```
type T_Tableau (Discriminant : T_Intervalle) is
```

```
record
```

```
    Champ_1 : Integer;
```

```
    Champ_2 : String(1..Discriminant);
```

```
end record;
```

```
Tab : T_Tableau(5);    -- contient un tableau de 5 elements
```

```
type T_Coefficients is array (T_Intervalle range  $\langle \rangle$ ) of Integer;
```

```
type T_Polynome (Degre : T_Intervalle) is    -- Pour traiter des polynomes
```

```
record
```

```
    -- Coefficients des termes
```

```
    Coefficients : T_Coefficients (T_Intervalle'First..Degre);
```

```
end record;
```

Discriminant (exemple)

```
type T_Matrice is array (Integer range <>,
                          Integer range <>) of Float;
```

-- une matrice de type T_Carrée est carrée !

```
type T_Carrée (Ordre : Positive) is
  record
    Mat : T_Matrice (1 .. Ordre, 1 .. Ordre);
end record;
```

```
Matrice_3 : T_Carrée (3);
```

```
subtype T_Carree_3 is T_Carree (3);
```

Agrégats, expressions et opérations

- En l'absence d'une valeur par défaut pour un discriminant (cf. 12.3), la déclaration d'un article (constante ou variable) à discriminant doit comporter une valeur pour chacun de ses discriminants, valeur donnée entre parenthèses ou par une valeur initiale (agrégat). Un tel article est dit **contraint**, par analogie avec la déclaration d'un tableau.

- $P(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$ avec $a_n \neq 0$ si $n \neq 0$

- **Sous-types articles à discriminants**

subtype identificateur **is** id_type_article (valeur_discr_1, ..., valeur_discr_N);

-- Pour utiliser des polynomes de degres 2 et 5

subtype T_Polynome_2 **is** T_Polynome (2);

subtype T_Polynome_5 **is** T_Polynome (5);

Quadratique : T_Polynome_2;

-- *Un polynome de degre 2*

Polynome_Degre_5 : T_Polynome_5;

-- *Un polynome de degre 5*

-- Pour afficher n'importe quel polynome

procedure Afficher (Polynome : **in** T_Polynome);

Agrégats, expressions et opérations

-- *Le polynome $3 - 2x + 5x^2$*

Polynome_1: T_Polynome (2) := (2, (3, -2, 5));

-- *Le polynome $x + 7x^4$ déclaré de deux manieres equivalentes*

Polynome_2: T_Polynome (4) := (4, (0, 1, 0, 0, 7));

Polynome_3: T_Polynome := (4, (0, 1, 0, 0, 7));

-- *Le polynome constant -6*

Polynome_4: **constant** T_Polynome (0) := (Degre => 0, Coefficients => (0 => -6));

-- *Le polynome $1+x+x^2+\dots+x^n$*

Polynome_5: T_Polynome := (N, (0..N => 1)); -- N entier et cf. 9.2.3

■ Les opérations:

- Affectation :=
- Passage en paramètre
- Comparaison = /=

Affectation

procedure Exemple **is**

subtype Intervalle **is** Integer **range** 0 .. 100;

type T_Coefficients **is array** (Intervalle **range** $\langle \rangle$) **of** Integer;

type T_Polynome (Degre : Intervalle) **is** -- Pour traiter des polynomes

record

Coefficients : T_Coefficients (Intervalle'First..Degre); -- *Coefficients des termes*

end record;

-- *Une constante et deux variables de ce type*

Deux_Plus_X : **constant** T_Polynome := (1, (2, 1));

Polynome_1 : T_Polynome (1) ;

Polynome_2 : T_Polynome := (Degre => 2, Coefficients => (2, -4, 1));

-- *Pour afficher n'importe quel polynome*

procedure Afficher (Polynome : **in** T_Polynome) **is begin ... end** Afficher;

begin -- *Exemple*

Polynome_1 := Deux_Plus_X; -- *Correct car memes*

Polynome_2 := (Polynome_2.Degre, (1, -2, 1)); -- *valeurs de*

Polynome_2 := T_Polynome'(2, (1, -2, 1)); -- *discriminant*

Afficher (Deux_Plus_X); -- *Affichage de deux polynomes*

Afficher (Polynome_2);

Polynome_2 := Polynome_1; -- *Provoque Constraint_Error (discriminants differents)*

end Exemple;

Valeurs par défaut des discriminants

- L'inconvénient majeur des articles à discriminants réside dans l'impossibilité de modifier la contrainte du discriminant. Le type d'un discriminant doit être de type discret ou accès
- En introduisant une valeur par défaut pour le discriminant lors de la déclaration du type article, ce type devient alors non contraint. On a cependant la possibilité de modifier les champs par la suite.
- La modification de la valeur d'un discriminant d'un article non contraint n'est possible que par une *affectation globale* de l'article, au moyen d'un agrégat.

Valeurs par défaut des discriminants

procedure *Exemple* **is** :

subtype T_Degre **is range** 0..100;

type T_Coefficients **is array** (T_Degre **range** <>) **of** integer;

type T_Polynome (Degre : T_Degre := 0) **is** --0 est la valeur par default

record

Coefficients : T_Coefficients(T_Degre'First.... T_Degre);

end record;

-- Une constante et une variable contrainte de ce type

Deux_Plus_X : **constant** T_Polynome(1,(2,1));

Polynome_1 : T_Polynome(1);

-- Deux variables de ce type non contraintes

Polynome_2 : T_Polynome := (Degre=>2, Coefficients=>(2, -4, 1));

Polynome_3 : T_Polynome;

begin

Polynome_1 := Deux_Plus_X ;

Polynome_2 := (Polynome_2.Degre, (1, -2, 1)) ;

Polynome_2 := T_Polynome'(2, (1, -2, 1)); -- qualification souhaitable

Polynome_3 := Polynome_1 ;

Polynome_3 := (Polynome_2.Degre - 1, Polynome_2.Coefficients(0..Polynome_2.Degre-1));

end *Exemple*;

Attribut Constrained

- Cette procedure normalise le polynome P , c'est-à-dire qu'elle assure
- que le polynome rendu dans P , de degre n , a son terme $a_n x^n$ non nul.
- Il faut cependant que le parametre effectif soit non contraint.

```

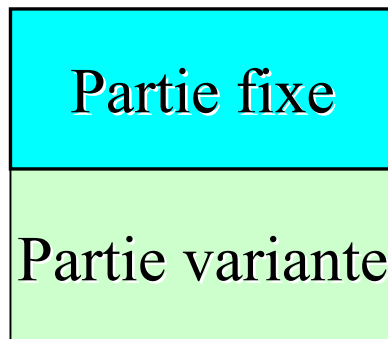
procedure Normaliser ( P : in out T_Polynome ) is
  Degre_Polynome : Natural := P.Degre;  -- Degre du polynome
begin -- Normaliser
  -- Normalisation possible?
  if not P'Constrained then
    -- Chercher le plus grand coefficient non nul
    while Degre_Polynome>0 and P.Coefficients(Degre_Polynome)=0 loop
      Degre_Polynome := Degre_Polynome - 1;
    end loop;
    P := ( Degre_Polynome,
          P.Coefficients ( Intervalle'First..Degre_Polynome ) );
  end if;
end Normaliser;
  
```

Articles à parties variantes

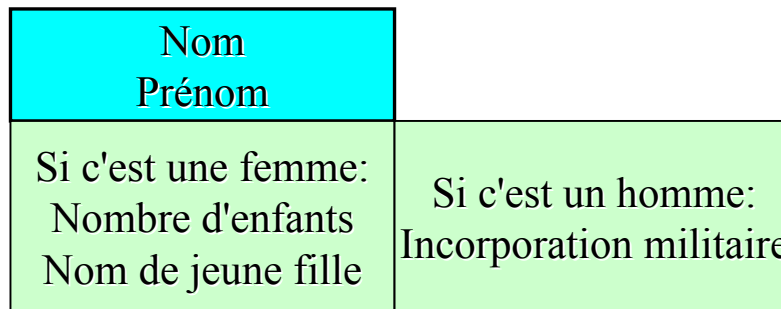
•Jusqu'à présent:



On va introduire:



Exemple une personne:

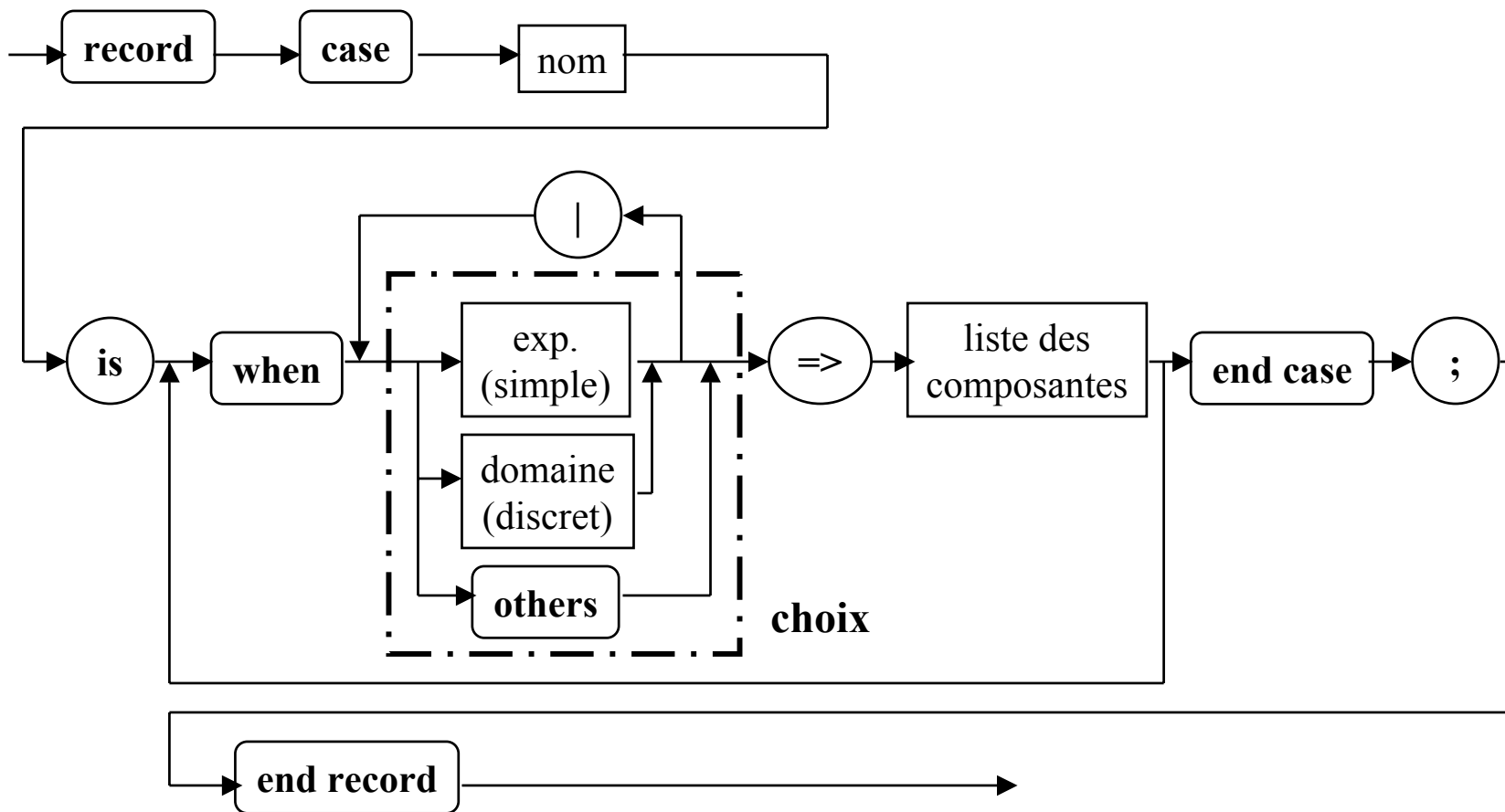


Partie Fixe

Partie Variante

Présentation d'un (sous-) type enregistrement à parties variantes

Une partie variante est une structure permettant de déclarer les champs particuliers de certains articles. La forme générale est :



Partie variante(Exemple)

```
type T_Genre is (Masculin, Feminin);
```

```
type T_Individu (Sexe : T_Genre) is
```

```
record
```

```
    Naissance : T_Date;
```

```
    case Sexe is
```

```
        when Masculin =>
```

```
            Barbu : Boolean;
```

```
        when Feminin =>
```

```
            Enfants : Integer;
```

```
    end case;
```

```
end record;
```

```
Pascal : T_Individu (Masculin);
```

```
subtype T_Femme is T_Individu (Sexe => Feminin);
```

Présentation d'un (sous-) type enregistrement à parties variantes (suite)

case discriminant is

when choix_1 => suite_de_declarations_de_champs_1;

when choix_2 => suite_de_declarations_de_champs_2;

when choix_3 => suite_de_declarations_de_champs_3;

...

when others => autre_suite_de_declarations_de_champs;

end case;

■ avec

- discriminant d'un type discret et déclaré comme discriminant d'article;
- les choix_n statiques, formés de valeurs ou d'intervalles séparés par des barres verticales, valeurs et bornes d'intervalle du type du discriminant;
- les suite_de_declarations_de_champs_n composées d'une ou de plusieurs déclarations de champs (éventuellement aucune);
- le mot réservé **others** qui représente toutes les autres valeurs possibles du discriminant.

Présentation d'un (sous-) type enregistrement à parties variantes (suite)

```
type TOrientation (Paysage, Portrait);
```

```
type TPeripherique is (Ecran, Imprimante) ;
```

```
type TStatus is (Libre, Occupe);
```

```
type TSortie(Unite : TPeripherique) is
```

```
  record
```

```
    Etat : TStatus ;                -- champs communs
```

```
    Vitesse : integer;
```

```
  case Unite is
```

```
    when Ecran =>    Format : array(1..80, 1..24) of character;
```

```
    when Imprimant => Format : array (1..132, 1..56) of character;
```

```
    Orientation : TOrientation;
```

```
    Nb_Copies : integer;
```

```
  end case;
```

```
end record;
```

```

Max : constant := 80;                -- Longueur maximum d'une ligne
type T_Genre_Fenetre is (Confirmation, De_Taille_Variable, Avertissement);
type T_Fenetre ( Genre : T_Genre_Fenetre := De_Taille_Variable ) is
  record
    Abscisse : Integer;                -- Position de la fenetre
    Ordonnee : Integer;
    Longueur : Integer;                -- Dimensions de la fenetre en pixels
    Largeur : Integer;
    case Genre is                      -- Partie variantes
      when Confirmation =>
        Texte_Affiche : String ( 1..Max );    -- Selon le texte affiche,
        Reponse_OK : Boolean;                 -- confirmer ou non
      when De_Taille_Variable =>
        Variation_X : Integer := 0;          -- Variations par rapport
        Variation_Y : Integer := 0;          -- aux dimensions originales
      when Avertissement =>
        Texte_Avertissement : String ( 1..Max );
    end case;
  end record;
Edition : T_Fenetre ( De_Taille_Variable );  -- Article contraint
Graphique : T_Fenetre;                       -- Article non contraint
Erreur_NT : T_Fenetre ( Avertissement );     -- Article contraint

```

Expressions, agrégats et opérations sur les articles à parties variantes

```

Edition : T_Fenetre ( De_Taille_Variable ) :=      -- Utilisation des
          ( De_Taille_Variable, 0, 0, 600, 400, 0, 0 ); -- valeurs par default

-- Graphique est non contraint malgre la valeur initiale
Graphique : T_Fenetre := ( De_Taille_Variable, 0, 0, 600, 400, 10, 20 );

-- Agregat tableau (cf. 9.2.3) pour le texte d'avertissement
Erreur_NT : T_Fenetre ( Avertissement ) :=
          ( Avertissement, 0, 0, 600, 400, "Profil inconnu" & (15..80=>' ') );

```

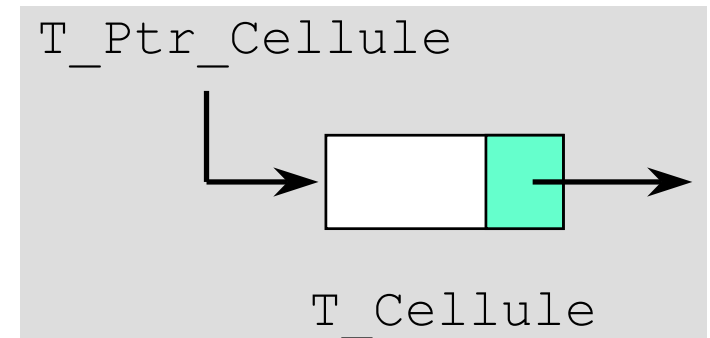
Sommaire

- Chapitre I : Présentation
- Chapitre II : Unités lexicales
- Chapitre III : Types et sous types
- Chapitre IV : Ordres (Sélection, cas, itération, ...)
- Chapitre V : Sous programmes
- Chapitre VI : Tableaux (array)
- Chapitre VII : Chaînes de caractères (String)
- Chapitre VIII : Articles (record)
- **Chapitre IX : Pointeur**
- Chapitre X : Fichiers (File)
- Chapitre XI : Paquetages simples (package)
- Chapitre XII : Généricité
- Chapitre XIII : Tâches

Pointeurs sur les objets

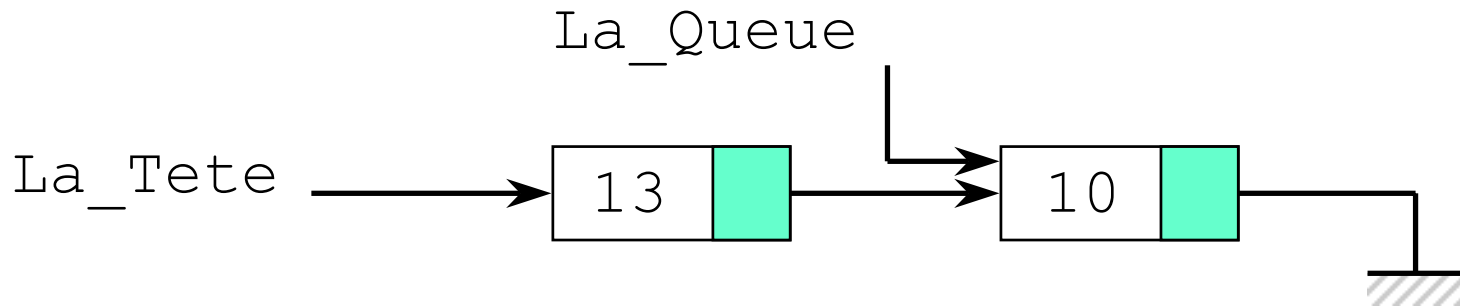
```
type T_Cellule;  
type T_Ptr_Cellule is access T_Cellule;
```

```
type T_Cellule is  
record  
    Contenu : Integer;  
    Ptr_Suivant : T_Ptr_Cellule;  
end record;
```



```
La_Tete, La_queue : T_Ptr_Cellule;
```

```
La_Tete := new T_Cellule'(13, La_Queue);  
La_Queue := new T_Cellule'(10, null);
```



Accès au contenu

```
La_Tete.Contenu           -- c'est 13  
La_Tete.Ptr_Suivant      -- c'est La_Queue
```

Copie les pointeurs et les contenus

```
La_Tete := La_Queue;      -- les pointeurs  
La_Tete.all := La_Queue.all; -- les contenus
```


Sommaire

- Chapitre I : Présentation
- Chapitre II : Unités lexicales
- Chapitre III : Types et sous types
- Chapitre IV : Ordres (Sélection, cas, itération, ...)
- Chapitre V : Sous programmes
- Chapitre VI : Tableaux (array)
- Chapitre VII : Chaînes de caractères (String)
- Chapitre VIII : Articles (record)
- Chapitre IX : Pointeur
- **Chapitre X : Fichiers (File)**
- Chapitre XI : Paquetages simples (package)
- Chapitre XII : Généricité
- Chapitre XIII : Tâches

Chapitre IX : Les Fichiers

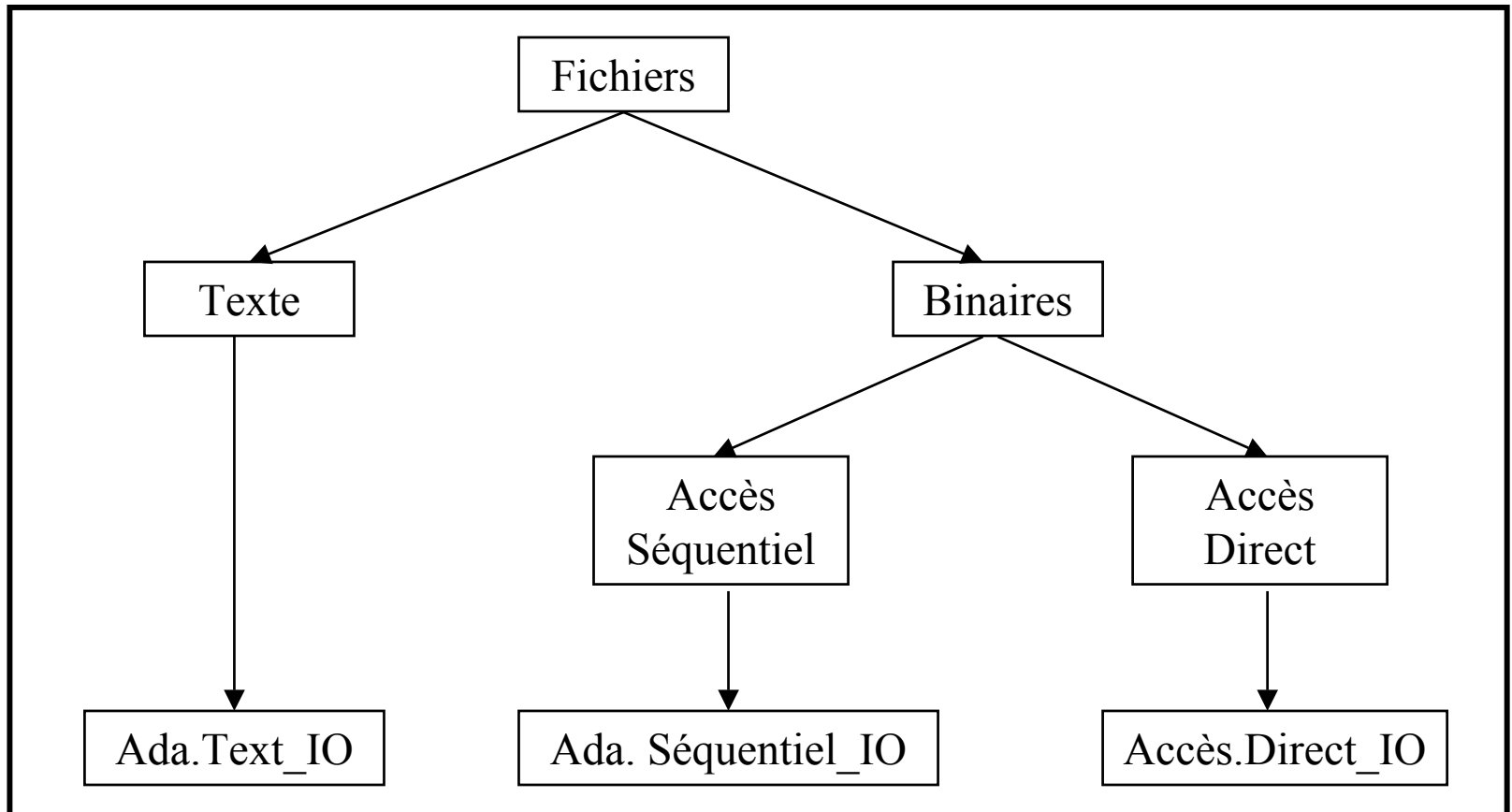
- **I. Introduction sur les différents types de fichiers**
 - I.1 Présentation des fichiers
- **II. Manipulation des fichiers texte**
 - II.1 Fichier texte
 - II.2 Manipulation des fichiers texte
 - II.3 Creation Ouverture Traitement et Fermeture de fichiers texte
 - II.4 Exemple d'utilisation d'un Fichier texte
 - II.5 Complément sur les fichiers texte
- **III. Manipulation des fichiers binaires**
 - III.1 Fichiers binaires
 - III.2 Creation Ouverture Traitement et Fermeture de fichiers binaires
 - III.3 Fichiers binaires séquentiels
 - III.4 Fichiers binaires à accès direct
 - III.5 Accès aux éléments dans un fichier binaire à accès direct
 - III.6 Remarques sur l'utilisation des fichiers binaires à accès direct
- **IV. Compléments sur l'utilisation des fichiers**
- **V. Exceptions lors de l'utilisation des fichiers**

I.1 Présentation des Fichiers

- Un fichier doit exister pour être utilisé, il faudra donc qu'il soit créé au moins une fois.
- Un fichier doit être ouvert avant utilisation et on ne peut ouvrir qu'un fichier qui a été préalablement été créé.
- Un fichier a un début et une fin. On indique la fin de fichier par un symbole de fin de fichier.
- Un fichier peut s'ouvrir :
(1) en mode : de *lecture*, (2) en mode d'*écriture* ou (3) *les deux*, selon le type de fichier.
- Il existe un curseur pour marquer la position courante dans un fichier.
- À l'ouverture, le curseur est au début du fichier ou à la fin, selon le mode d'ouverture.
- Toutes les opérations de lecture et d'écriture dans un fichier se font par rapport à ce curseur.
- Une lecture se fait à la position courante et le curseur est déplacé sur la prochaine donnée à lire.
- Une écriture se fait à la position courante et le curseur est déplacé sur la prochaine donnée à écrire.
- On ne doit jamais tenté de lire si le curseur est à vis à vis le symbole de fin de fichier.
- Il existe des fonctions pour savoir si le curseur est vis à vis le symbole de fin de fichier.
- On doit fermer un fichier lorsqu'on a terminé de s'en servir.

I.1 Présentation des Fichiers

- Les fichiers sont universellement utilisés pour communiquer des informations d'un programme à un autre et/ou pour communiquer des informations à un utilisateur, et/ou pour sauvegarder des données sur un support.
- Les 2 grandes catégories de fichiers sont les fichier *texte* et les fichiers *binaires*.



I.1 Présentation des Fichiers

- Les fichiers *texte* sont les plus simples à utiliser. Ces fichiers contiennent du texte brut, reparti en ligne. Chaque ligne se termine par un marqueur de fin de ligne (FL) constitué de caractères spéciaux (retour chariot, saut de ligne). Un fichier de texte se termine par un symbole appelé "fin de fichier"(FF)
 - Stocker du texte simple
 - Écrire des fichiers de configuration (fichier.ini)
 - Générer des fichiers lisibles par d'autres applications (HTML, éditeur ADA, RTF)

- Les fichiers binaire contiennent des suites binaires donc non lisibles à l'oeil
 - Les fichiers *séquentiels* ou fichier *typé*, servent à stocker un certain nombre d'*enregistrements* d'un type donné, et ce de façon automatisée. Tous les types de données ne peuvent pas être stockés dans ce type de fichier (classes, pointeurs ou tableaux dynamique). Ce genre de fichier est surtout utilisé pour stocker une liste des éléments de type record.
 - ✓ Point fort => facilité d'utilisation
 - ✓ Point faible => aucune liberté, pas d'utiliser de pointeurs et tableaux dynamiques

 - Les fichiers à *accès direct* sont appelé fichiers universels. Ils peuvent contenir n'importe quel type de données (pointeurs, tableau dynamique...).

- Traitement d'un fichier:

- Préparation, ouverture
 - Open / Create
 - ✓ Etablit le lien entre la variable fichier du programme et le fichier externe

 - Traitement proprement dit: Lecture / Ecriture
 - ✓ Get / Put Read / Write ...
 - Réalise le traitement effectif désiré sur les enregistrements du fichier.

 - Terminaison, fermeture du fichier:
 - ✓ Close
 - Vide le tampon associé au fichier si nécessaire.
 - Coupe la liaison: variable de fichier du programme - Fichier externe.

II.1 Fichier Texte

L'ordinateur a besoin d'un espace mémoire pour chaque fichier qu'on appelle "tampon" ou en anglais "buffer". Ce tampon peut contenir plusieurs caractères et est rempli chaque fois qu'il est vide. Le mécanisme est que si une instruction de lecture est faite dans un fichier de texte et que le tampon est vide, il sera rempli par les caractères du fichier et ensuite l'affectation se fera à partir du tampon.

Pour associer un tampon avec un fichier, il faut déclarer une variable qui servira de tampon.

```
Fichier_Texte : Ada.Text_IO.File_Type;
```

C'est à l'ouverture du fichier que le lien se fera entre le fichier réel et le tampon en mémoire.

```
Ada.Text_IO.Create(Fichier_Texte, Ada.Text_IO.Out_File, "a :\ Data.txt");
```

Cette dernière instruction, créera un fichier appelé "Data.txt" sur l'unité de disquette "a:", et associera un tampon appelé "Fichier_Texte". Dorénavant, toutes les instructions dans ce fichier se feront via le tampon.

```
Ada.Text_IO.Put(Fichier_Texte,"salut vous autres");
```

À la fin du programme il faut couper le lien entre le fichier et le tampon. Pour ce faire il faut utiliser la procédure close:

```
Ada.Text_IO.Close(Fichier_Texte);
```

II.2 Manipulation des fichiers texte

```

with Ada.Text_IO ; use Ada.Text_IO
-- ...
procedure Exemple is
    Original : File_Type ; -- Deux variables fichier texte
    Copie    : File_Type ;

    procedure Dupliquer (Source : in File_Type ; Destination : in File_Type) is
    begin
        .....
    end Dupliquer;

begin -- Exemple
    .....
end Exemple;

```

Remarque :

- La variable de fichier, objet (variable) interne au programme ADA
- Le fichier externe

II.3 Création, Ouverture Traitement et Fermeture de fichiers texte

- **Création** --nouveau fichier

```
procedure Create ( File : in out File_Type;  --type de fichier, texte ou binaire
                  Mode : in File_Mode := out_File; --in_File ou Append_File
                  Name : in String := "";      --nom du fichier externe
                  Form  : in string := "");    --parametre fixant les proprietes...
                                          --exemple : les droits d'accès
```

- **Ouverture** --fichier existant

```
procedure Open ( File : in out File_Type;
                Mode : in File_Mode;
                Name : in String;
                Form  : in string := "");
```

- **Traitement**

Lecture, Ecriture, ..(Put, Get , Put_Line, Get_Line, New_Line, Skip_Line)

- **Ferméture** --suppression de l'association entre la variable fichier et le fichier externe

```
procedure Close (File : in out File_Type);
```

II.4 Exemple 1 d'utilisation d'un Fichier texte

```

with Ada.Text_IO;           use Ada.Text_IO ;
with Ada.Integer_Text_IO;   use Ada.Integer_Text_IO ;
with Ada.Float_Text_IO;    use Ada.Float_Integer_Text_IO ;

procedure Exemple is
    Buffer : File_Type ;           -- Une variable fichier texte
    Titre : string(1..10);       -- Une variable chaine de caracteres
    Lettre : Character ;         -- Une variable caractere
    Nb_Entier : Integer;        -- Une variable entiere
    Nb_Reel : Float ;           -- Une variable reelle
    L : Natural;                -- Pour un appel correct a Get_Line
begin
    -- Creation et ouverture du fichier a ecrire appeler texte.txt
    Create(File => Fichier_Traite, Name => "texte.txt");
    Put_Line(Fichier_Traite, "Titre"); -- Ecriture sur une ligne du mot Titre
    Put(Fichier_Traite, 'O');         -- Ecriture d'un caractere
    Put(Fichier_Traite, 123);        -- Ecriture d'un entier
    Put(Fichier_Traite, 'N');       -- Ecriture d'un caractere
    New_Line (Fichier_Traite);      -- Passage a la ligne
    Put(Fichier_Traite, 3.14);      -- Ecriture d'un reel
    New_Line (Fichier_Traite);     -- Passage a la ligne
    Close(Fichier_Traite);         -- Fermeture du fichier

```

.....

II.4 Exemple 1 d'utilisation d'un Fichier texte (suite)

```

-- suite de la procedure precedante Exemple

-- Ouverture du meme fichier, cette fois-ci en lecture
Open(File => Fichier_Traite, Mode => In_File, Name => "texte.txt");
Get_Line( Fichier_Traite, Titre, L);  -- Lecture du titre (L vaut 5)
Get(Fichier_Traite, Lettre);          -- Lecture d'un caractere (O)
Get(Fichier_Traite, Nb_Entier);       -- Lecture d'un entier(123)
Get(Fichier_Traite, Lettre);          -- Lecture d'un caractere (N)
Skip_Line (Fichier_Traite);           -- Passage a la ligne
Get(Fichier_Traite, Nb_Reel);         -- Lecture d'un reel
Skip_Line (Fichier_Traite);           -- Passage a la ligne
Close(Fichier_Traite);                -- Fermeture du fichier
end Exemple ;

```

.....

Le fichier texte.txt créé par la procédure Exemple

Titre

O 123N

3.14000E+00

II.4 Exemple 2 d'utilisation d'un Fichier texte

```
function End_Of_Line (File : in File_Type) return Boolean;
```

```
function End_Of_File (File : in File_Type) return Boolean;
```

```
-----
```

```
with Ada.Text_IO ; use Ada.Text_IO ;
```

```
procedure Exemple is
```

```
    Fichier_Traite : File_Type ;           -- Une variable fichier texte
```

```
    Lettre : Character ;                   -- Une variable caractere
```

```
begin
```

```
    -- Ouverture du fichier a lire
```

```
    Open(File => Fichier_Traite, Mode => In_File, Name => "texte.txt");
```

```
    while not End_Of_File( Fichier_Traite) loop           -- Fin de fichier ?
```

```
        while not End_Of_Line( Fichier_Traite) loop     -- Fin de ligne ?
```

```
            Get(Fichier_Traite, Lettre);                  -- Lecture d'un caractere
```

```
            .....
```

```
        end loop;
```

```
        Skip_Line (Fichier_Traite);                       -- Passage a la ligne
```

```
    end loop;
```

```
    Close(Fichier_Traite);                                -- Fermeture du fichier
```

```
end Exemple ;
```

II.5 Complément sur les fichiers texte

procedure Get_Immediate(Item : **out** Character);

procedure Get_Immediate(Item : **out** Character;
Available : **out** Boolean);

- Item le caractère lu immédiatement (clavier ou fichier);
- Available qui indique s'il existe un caractère à lire immédiatement
- S'il n'y a pas de caractère à lire, la première forme fait attendre, la deuxième retourne False

procedure Look_Ahead (Item : **out** Character;
End_Of_Line : **out** Boolean);

- permet d'obtenir une copie du caractère à lire;
- Item le caractère obtenu si End_Of_Line est faux;
- End_Of_Line est vrai si la fin de ligne est atteinte. Dans ce cas Item n'a pas de valeur définie.

III. Fichiers binaires

Le contenu d'un fichier binaire peut être de tout type (élémentaire, tableau, article, tableau d'articles..). Il n'y a que 2 instructions qui permettent de manipuler le contenu d'un fichier binaire en Ada et c'est READ(lecture) et WRITE(écriture).

Pour pouvoir utiliser un fichier binaire, il faut déterminer son mode d'accès :

- Accès séquentielle => nous devons importer ADA.SEQUENTIAL_IO.
- Accès direct => nous devons importer ADA.DIRECT_IO.

Par la suite, nous devons spécifier de quel type est le fichier et cela se fait par l'instanciation de paquetage (*package générique*).

```
ex; PACKAGE ES_Fic_Etud IS NEW ADA.SEQUENTIAL_IO(T_Etud);
```

Cette dernière instruction nous donne tous les outils pour pouvoir utiliser un fichier binaire de type "T_Etud" en accès séquentiel. Naturellement, le type "T_Etud" doit avoir préalablement été défini. À partir du moment où le paquetage est défini, n'importe quel référence à un type, une procédure ou une fonction qui provient de ADA.SEQUENTIAL_IO doit être utiliser avec le nom du paquetage en préfixe.

```
ex: Un_Fichier_Binaire : ES_Fic_Etud.File_Type;
     ES_Fic_Etud.Open(Un_Fichier_Binaire, ES_Fic_Etud.IN_FILE, "Unfichier.bin");
```

III. Fichiers binaires

```

with Ada.Sequential_IO;           -- Pas de use sur un paquetage qui
with Ada.Direct_IO;             -- est generique (cf. ...)
procedure Exemple is
    type T_Mois_De_L_Annee is (Janvier, Fevrier, Mars, Avril, Mai, Juin, Juillet, Aout,
                                Septembre, Octobre, Novembre, Decembre );

    type T_Date is record
        Jour   : Integer;
        Mois   : T_Mois_De_L_Annee;
        Annee  : Integer;
    end record;

    -- Pour utiliser des fichiers binaires sequentiels d'entiers
    package Entiers_Seq_IO is new Ada.Sequential_IO ( Integer );    use Entiers_Seq_IO;
    -- Pour utiliser des fichiers binaires directs de dates
    package Dates_Dir_IO is new Ada.Direct_IO ( T_Date );        use Dates_Dir_IO;
    Fichier_Entiers : Entiers_Seq_IO.File_Type;    -- Deux variables fichiers
    Fichier_Dates   : Dates_Dir_IO.File_Type;     -- Prefixe necessaire
    -- Lit le fichier d'entiers Mesures
    procedure Lire (Mesures : in Entiers_Seq_IO.File_Type) is ... end Lire;
    -- Ecrit le fichier de dates Dates
    procedure Ecrire (Dates : in Dates_Dir_IO.File_Type) is ... end Ecrire;
begin -- Exemple

```

III.2 Création, Ouverture Traitement et Fermeture de fichiers binaires

■ Création

```

procedure Create (File : in out File_Type;
                  Mode : in File_Mode := ...; -- (Seq : Out_File, In_File, Append_File...
                  Name : in String := "";      -- Bin : InOut_File ,In_File, Out_File)
                  Form : in String := "" );    -- (Valeur par défaut du parametre Name...
                                                -- permet de créer un fichier temporaire)

```

■ Ouverture

```

procedure Open ( File : in out File_Type;
                 Mode : in File_Mode;
                 Name : in String;
                 Form : in String := "" );

```

■ Traitement Accès aux éléments : fichier binaire séquentiel

```

procedure Read ( File : in File_Type; Item : out Element_Type );
procedure Write ( File : in File_Type; Item : in Element_Type );

```

■ Fermeture

```

procedure Close ( File : in out File_Type );

```


III.3 Fichiers binaires séquentiels

with Ada.Sequential_IO;

...

procedure Exemple **is**

-- Pour utiliser des fichiers binaires séquentiels formes de dates,

-- les types sont declares comme dans l'exemple precedant

package Dates_Seq_IO **is new** Ada.Sequential_IO (T_Date); **use** Dates_Seq_IO;

Fichier_Dates : File_Type; *-- Une variable fichier binaire*

Date : T_Date; *-- Une date réelle*

begin -- Exemple

-- Creation et ouverture du fichier a ecrire

Create (File => Fichier_Dates, Name => "dates.dat");

Date := (1, Avril, 1958);

Write (Fichier_Dates, Date); *-- Ecriture de quelques dates*

Write (Fichier_Dates, (26, Decembre, 1964));

Write (Fichier_Dates, (7, Septembre, 1991));

Write (Fichier_Dates, (28, Juillet, 1998));

Close (Fichier_Dates); *-- Fermeture du fichier*

-- Ouverture du même fichier pour le relire

Open (File => Fichier_Dates, Mode => In_File, Name => "dates.dat");

Read (Fichier_Dates, Date); *-- Lecture des quatre dates dans*

Read (Fichier_Dates, Date); *-- l'ordre selon lequel elles*

Read (Fichier_Dates, Date); *-- viennent d'être écrites*

Read (Fichier_Dates, Date);

Close (Fichier_Dates); *-- Fermeture du fichier*

III.3 Fichiers binaires séquentiels

```
with Ada.Sequential_IO;
```

```
----
```

```
procedure Exemple is
```

```
    -- Pour utiliser des fichiers binaires séquentiels formes de dates,
```

```
    -- les types sont declares comme dans l'exemple précédant
```

```
package Dates_Seq_IO is new Ada.Sequential_IO ( T_Date );
```

```
use Dates_Seq_IO;
```

```
Original : File_Type;           -- Deux variables fichiers binaires séquentiels
```

```
Copie : File_Type;             -- On ne peut pas faire Original := Copie
```

```
Date : T_Date;                 -- Une variable pour la copie d'un element
```

```
begin -- Exemple
```

```
    -- Ouverture du fichier a lire et creation de la copie
```

```
Open ( File => Original, Mode => In_File, Name => "dates.dat" );
```

```
Create ( File => Copie, Name => "copie de dates.dat" );
```

```
while not End_Of_File ( Original ) loop    -- Fin de fichier?
```

```
    Read ( Original, Date );                -- Lecture d'une date
```

```
    Write ( Copie, Date );                  -- Ecriture d'une date
```

```
end loop;
```

```
Close ( Original );                        -- Fermeture des fichiers
```

```
Close ( Copie );
```

```
end exemple;
```

III.4 Fichiers binaires à accès direct

-- *Quelques déclarations appartenant au paquetage Ada.Direct_IO*

type Count **is range** 0 .. *implementation_defined*;

subtype Positive_Count **is** Count **range** 1 .. Count'Last;

procedure Read (File : **in** File_Type; Item : **out** Element_Type);

procedure Read (File : **in** File_Type; Item : **out** Element_Type; From : **in** Positive_Count);

procedure Write (File : **in** File_Type; Item : **in** Element_Type);

procedure Write (File : **in** File_Type; Item : **in** Element_Type; To : **in** Positive_Count);

Remarque : A chaque lecture ou écriture, l'indexe est augmenté de 1

procedure Set_Index (File : **in** File_Type; To: **in** Positive_Count);

function Index (File : **in** File_Type) **return** Positive_Count;

III.4 Traitement d'un fichier binaire à accès direct

with Ada.Direct_IO;

procedure Exemple **is**

-- Pour utiliser des fichiers binaires a acces direct formes de dates

-- les types sont declares comme dans l'exemple précédant

package Dates_Dir_IO **is new** Ada.Direct_IO (T_Date);

use Dates_Dir_IO;

Original : File_Type; *-- Deux variables fichiers a acces direct*

Copie : File_Type; *-- Une variable pour la copie d'un element*

begin *-- Exemple*

-- Ouverture du fichier a lire et creation de la copie

Open (File => Original, Mode => In_File, Name => "dates.dat");

Create (File => Copie, Name => "copie de dates.dat");

-- Copier tous les elements

for No_Element_Courant **in reverse** 1 .. Size (Original) **loop**

-- Placer l'index sur l'element No_Element_Courant du fichier lu

Set_Index (Original, No_Element_Courant);

Read (Original, Date); *-- Lecture d'une date*

Write (Copie, Date); *-- Ecriture d'une date*

end loop;

Close (Original); *-- Fermeture des fichiers*

Close (Copie);

end Exemple;

IV. Compléments sur l'utilisation des fichiers (texte et binaires)

procedure Delete (File : **in out** File_Type);

« Ferme le fichier et l'efface de la mémoire secondaire »

procedure Reset (File : **in out** File_Type);

« Rembobine dans le cas du mode lecture ou écriture »

procedure Reset (File : **in out** File_Type; Mode : **in** File_Mode);

« Repositionne au dernier élément dans le cas du mode adjonction »

function Is_Open (File : **in** File_Type) **return** Boolean;

« Vérifier si le fichier est ouvert ou non »

function Mode (File : **in** File_Type) **return** File_Mode;

« Informe sur le mode d'ouverture du fichier »

function Name (File : **in** File_Type) **return** String;

« Informe sur le nom du fichier externe connecté »

V. Exceptions lors de l'utilisation des fichiers (texte et binaires)

Toutes les opérations de traitement des fichiers ou de leurs éléments peuvent générer des exceptions. Ces exceptions sont toutes déclarées dans le paquetage `Ada.IO_Exception`

- **Status_Error** le fichier est ouvert alors qu'il devrait être fermé ou vice-versa;
- **Mode_Error** le fichier n'a pas le bon mode, par exemple `In_File` au lieu de `Out_File`;
- **Name_Error** une erreur est apparue dans le nom externe de `Create` ou `Open`;
- **Use_Error** diverses raisons (!); paramètre `Form` inacceptable, ordre d'impression sur un périphérique d'entrée etc;
- **Device_Error** le dispositif physique est en panne ou n'est pas connecté;
- **End_Error** tentative de lecture au-delà de la fin de fichier;
- **Data_Error** `Read` ou `Get` ne peut pas interpréter les données comme valeurs du type désiré;
- **Layout_Error** un problème de formatage dans `Ada.Text_IO`, ou bien `Put` écrit trop de caractères dans une chaîne.

Sommaire

- Chapitre I : Présentation
- Chapitre II : Unités lexicales
- Chapitre III : Types et sous types
- Chapitre IV : Ordres (Sélection, cas, itération, ...)
- Chapitre V : Sous programmes
- Chapitre VI : Tableaux (array)
- Chapitre VII : Chaînes de caractères (String)
- Chapitre VIII : Articles (record)
- Chapitre IX : Pointeur
- Chapitre X : Fichiers (File)
- **Chapitre XI : Paquetages simples (package)**
- Chapitre XII : Généricité
- Chapitre XIII : Tâches

Chapitre X : Paquetage simple

- X.1 Introduction
 - X.1.1 Modularité
 - X.1.2 Présentation d'un paquetage
 - X.1.3 Exemple
 - X.1.2 Appel d'une action (procédure)
 - X.1.3 Appel d'une fonction
 - X.1.4 Propriétés des paquetages

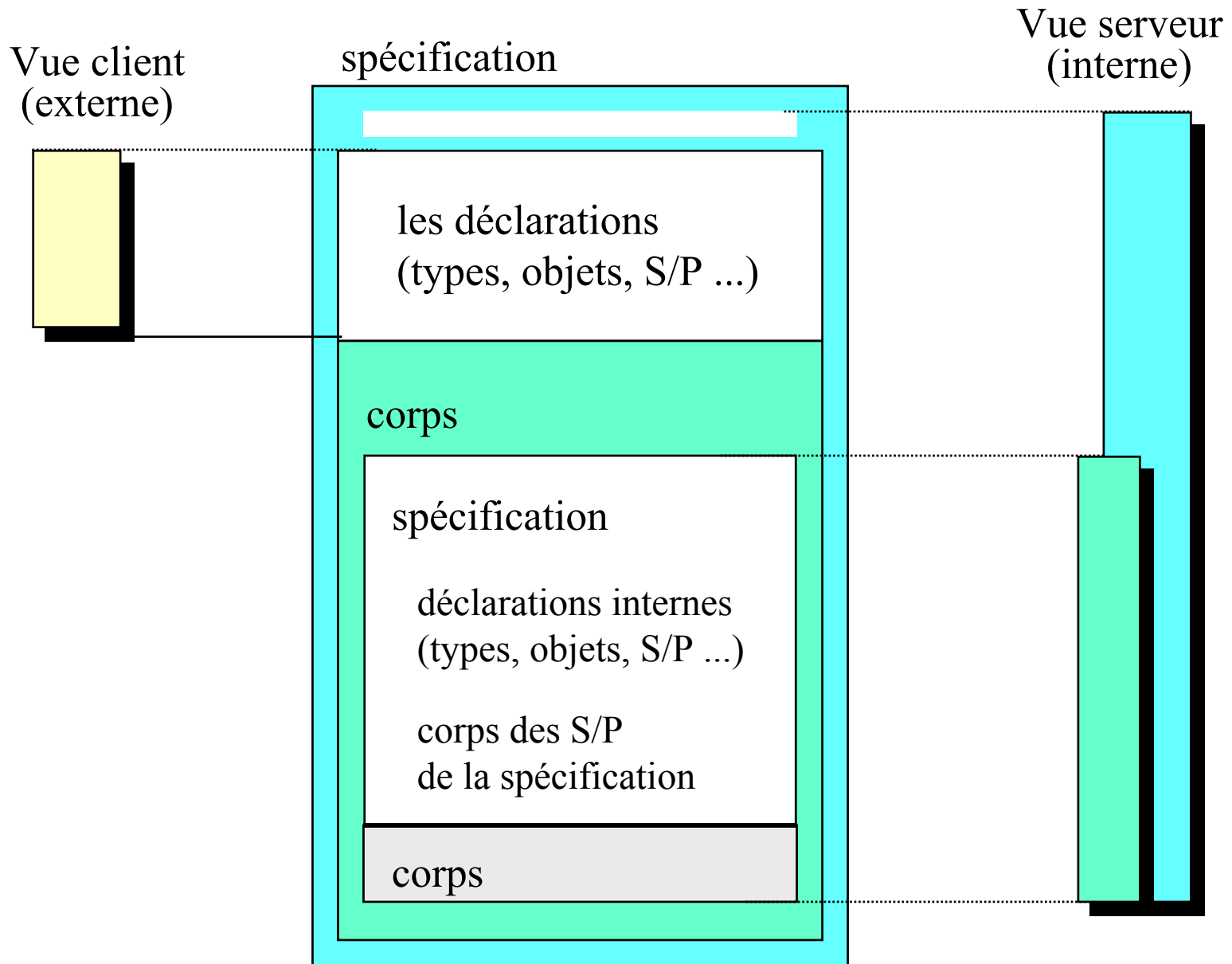
- X.2 Mise en place
 - X.2.1 Spécification de paquetage
 - X.2.2 Définition du corps de paquetage
 - X.2.3 Utilisation de paquetage : Clause use

- X.3 Exemple de programmation modulaire
 - X.3.1 Paquetage maths (spécification, corps, utilisation)
 - X.3.2 Nombres rationnels (spécification, corps et utilisation)
 - X.3.3 Paquetage nombres complexes
 - X.3.4 Utilisation du paquetage complexe avec la clause **use type**
 - X.3.5 Partie privée

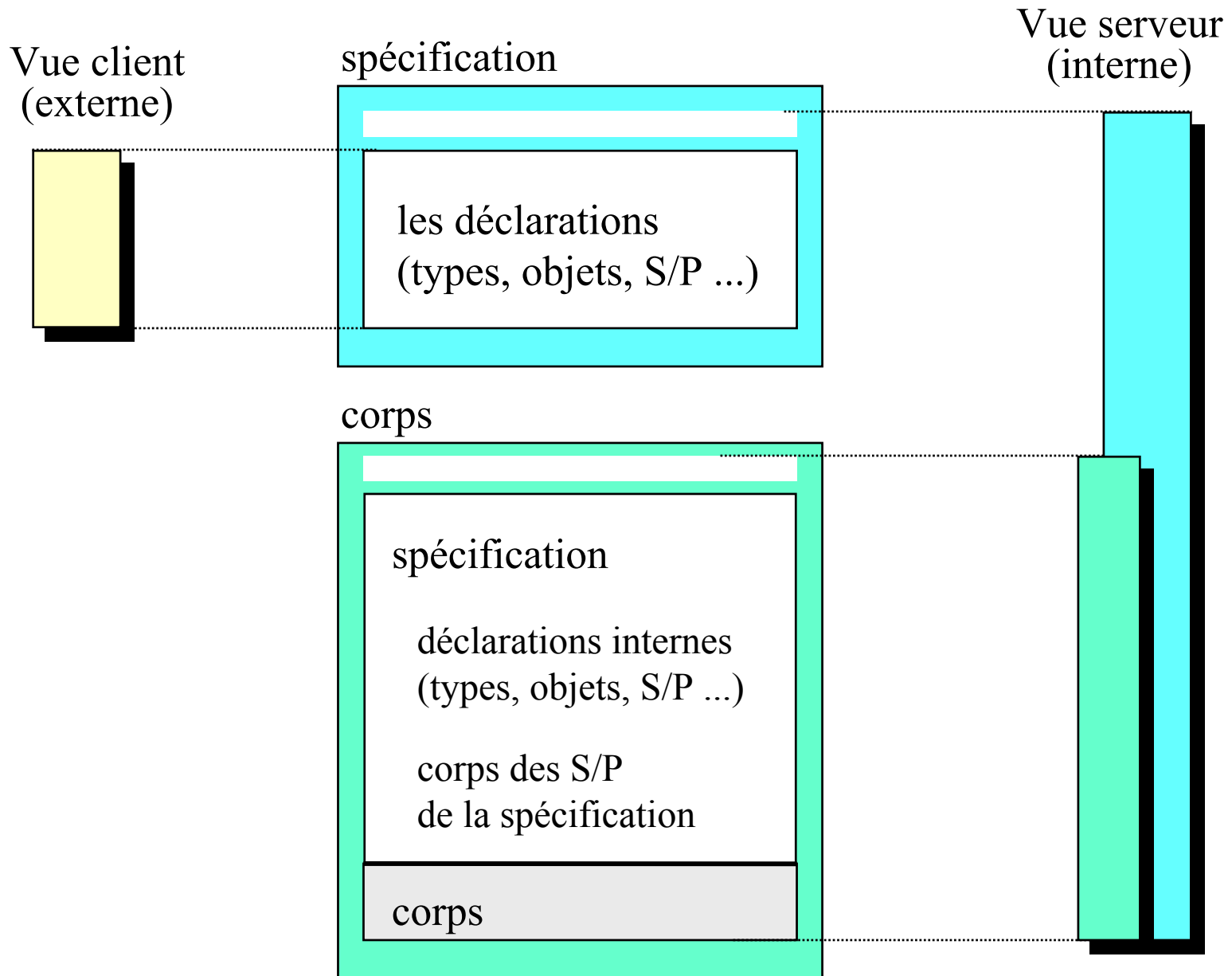
X.1.1 Modularité (compilation séparée), abstraction de données

- Présentation d'un paquetage : il est composé de deux parties :
 - spécification
 - ✓ partie visible
 - ✓ interface avec le monde extérieur, le client
 - ✓ contient les déclarations (types, objets, S/P ...)
 - corps (facultatif)
 - ✓ partie cachée
 - ✓ réalisation du service, le serveur
 - ✓ contient le corps des S/P de la spécification plus des déclarations et des S/P internes
- La spécification et le corps peuvent se trouver :
 - dans le même fichier
 - dans des fichiers séparés

X.1.2 Présentation d'un paquetage (même fichier)



X.1.2 Présentation d'un paquetage (fichier séparé)



X.1.3 Exemple

procedure Exemple **is**

Partie
visible

```
package Nb_Complexe is
  type T_Complexe is ...
  function Plus (X,Y : T_Complexe) return T_Complexe;
end Nb_Complexe;
```

Partie
cachée

```
package body Nb_Complexe is
```

```
Valeur_Initiale : T_Complexe ...
```

```
function Plus ... is
```

```
end Nb_Complexe;
```

```
Nb_A, Nb_B, Nb_C : Nb_Complexe.T_Complexe;
```

```
use Nb_Complexe;
```

```
begin -- Exemple
```

```
  Nb_C := Plus (Nb_A, Nb_B);
```

```
end Exemple;
```

X.1.3 Exemple (suite)

```

with Nb_Complexe;
procedure Exemple is
    Nb_A, Nb_B, Nb_C : Nb_Complexe.T_Complexe;
begin -- Exemple
    Nb_C := Nb_Complexe.Plus(Nb_A, Nb_B);
end Exemple;

```

```

with Nb_Complexe; use Nb_Complexe;
procedure Exemple is
    Nb_A, Nb_B, Nb_C : T_Complexe;
begin -- Exemple
    Nb_C := Plus(Nb_A, Nb_B);
end Exemple;

```

X.1.4 Appel d'une action

```
with NomPackage;                                -- en entete
```

```
...
```

```
NomPackage.NomProcedure( liste param effectifs );
```

Ou

```
with NomPackage;                                -- en entete
```

```
use NomPackage;
```

```
...
```

```
NomProcedure(liste param effectifs);
```

X.1.5 Appel d'une fonction

```
with NomPackage;                                -- en entete
```

```
...
```

```
A := NomPackage.NomFonction(liste param effectifs);
```

Ou

```
with NomPackage;                                -- en entete
```

```
use NomPackage;
```

```
...
```

```
A := NomFonction(liste param effectifs);
```

- **Modularité : Un package regroupe un ensemble de fonctions, procédures et types qui sont liés entre eux.**

Ceci est équivalent à la notion d'unité en Pascal ou à la notion de projet en C++

- **Réutilisation du code**

- **Lisibilité du code**

- **ADA impose une unique entité (fonction, procédure ou package) par fichier.**

➔ Obligation d'écrire des packages

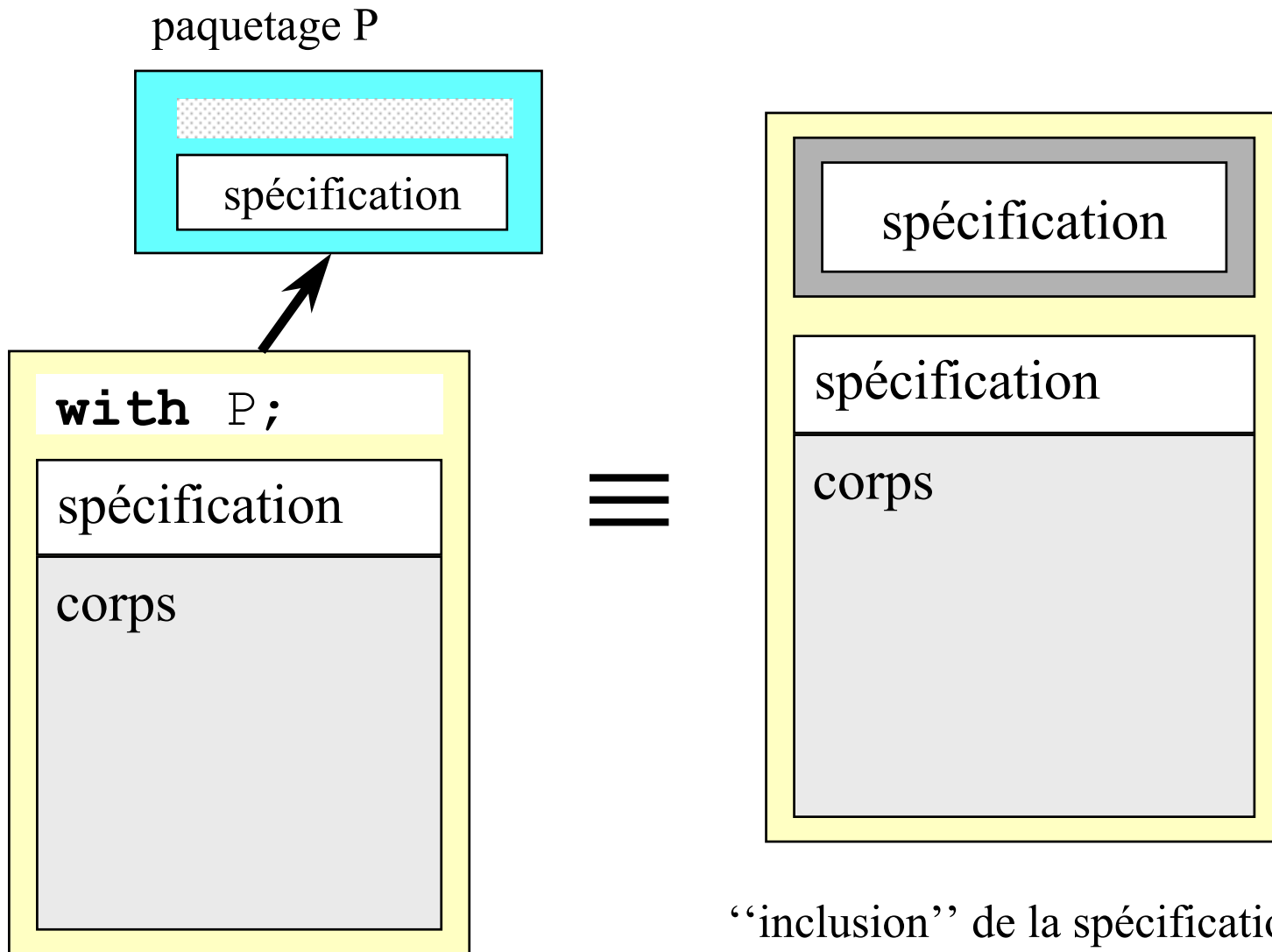
■ La clause **with**

- appel à une unité de bibliothèque
- dépendance directe à une unité de bibliothèque
- placée dans la partie déclarative
- visible aussi dans le corps

■ La clause **use**

■ Visibilité

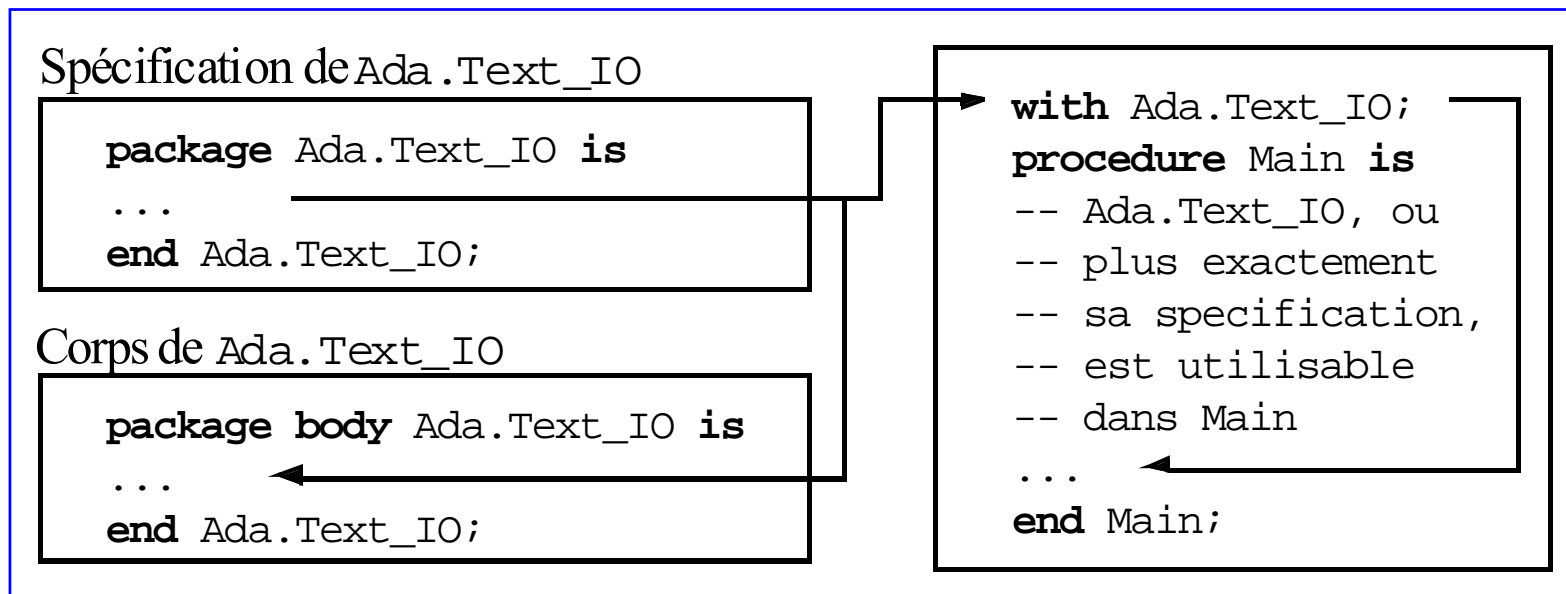
Clauses **with** et **use** : “inclusion” de la spécification.



“inclusion” de la spécification.

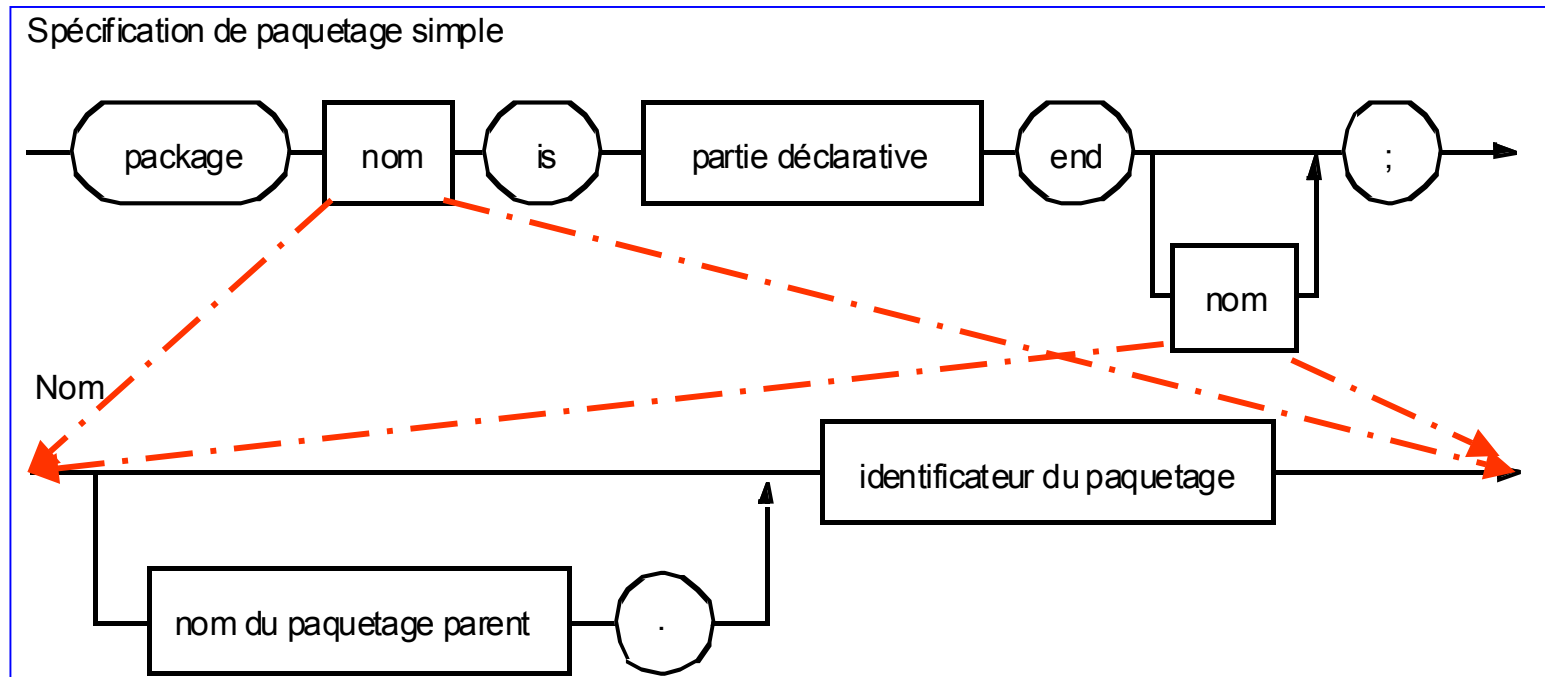
X.2 Mise en en place

- La **spécification** regroupe des déclarations visibles et utilisables dans et hors du paquetage .
- Le **corps** (*body*) englobe des éléments connus uniquement à l'intérieur du paquetage.
- Les déclarations utilisables à l'extérieur du paquetage sont appelés **exportées**.
- La durée de vie des variables déclarées dans un paquetage est celle du paquetage lui-même. Elles sont parfois appelées **variables rémanentes**.
- Les éléments choisis pour faire partie d'un paquetage doivent toujours tendre à constituer un tout cohérent.



X.2.1 Specification de paquetage

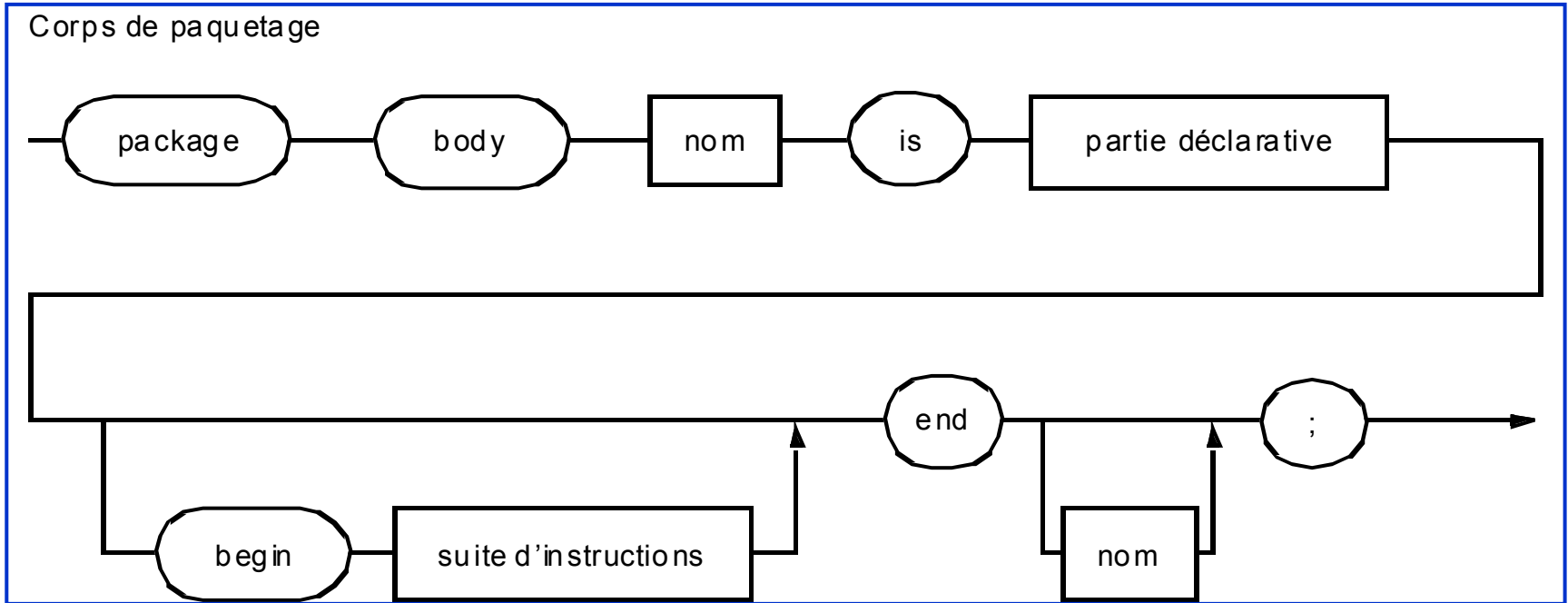
■ Structure d'une spécification de paquetage



- La spécification d'un paquetage peut contenir n'importe quelle déclaration sauf des corps. Il peut être de simples déclarations de type
exemple : Ada.Characters.Latin_1
Ada.IO_Exceptions

X.2.2 Définition du corps de paquetage

■ Code d'initialisation d'un paquetage:



package body Nom **is**

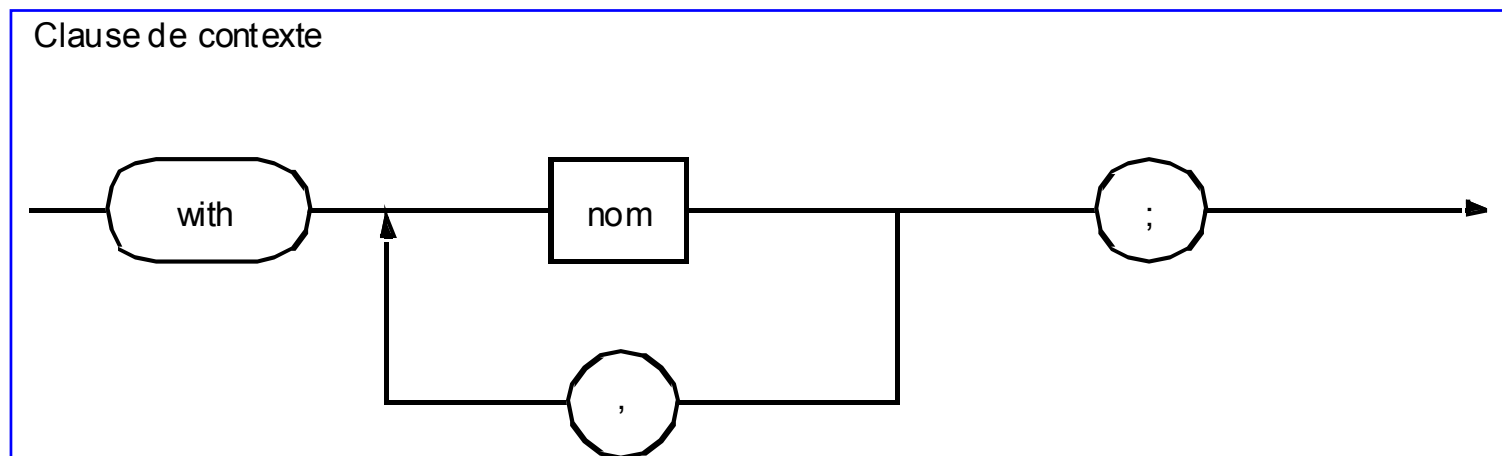
... -- *Declarations locales et corps*

begin

... -- *Instructions: code d'initilisation.*

end Nom;

X.2.3 Utilisation de paquetage : Clause use



- Un identificateur déclaré dans une spécification de paquetage est rendu directement visible par une clause **use** pourvu que le même identificateur ne figure pas dans un autre paquetage mentionné avec une clause **use**, et que cet identificateur ne soit pas déjà directement visible. Si ces conditions ne sont pas remplies, le préfixage est indispensable.
- Si tous les identificateurs en présence sont des sous-programmes ou des valeurs énumérées, alors ils se surchargent et deviennent tous directement visibles. Il est donc possible que des cas d'ambiguïté surviennent, qu'il faudra éliminer par exemple en préfixant chaque identificateur ambigu par un nom de paquetage.

Exemple File_type dans les paquetages

```
Ada_Text_IO , Ada.Sequential_IO , Ada.Direct_IO;
```

X.3.1 Paquetage maths (spécification)

Exemple : le fichier maths.ads

- maths.ads
- Spécification du package maths : diverses fonctions mathématiques
- Auteur :

- Spécification du package maths

package maths **is**

procedure Permuter(X,Y : **in out** Integer); *-- permute les valeurs de X et Y*

function Max(A,B : **in** Integer) **return** Integer; *-- renvoie le max de A et B*

private

function Renvoie(A : **in** integer) **return** Integer; *-- renvoie la valeur en entree*

end maths;

X.3.1 Paquetage maths (corps)

Exemple : début du fichier maths.adb

-- maths.adb : Corps du package maths

-- Auteur :

package body maths is

procedure Permuter(X,Y : **in out** Integer) **is** -- *permuter les valeurs de X et Y*

 Tmp : Integer; -- *sauvegarde temporaire de X*

begin

 Tmp := X;

 X := Y;

 Y := Tmp;

end Permuter;

-- voir la suite dans le slide suivant

X.3.1 Paquetage maths (corps)

Exemple : fin du fichier maths.adb

```
function Max(A, B : in Integer) return Integer is
```

```
  -- renvoie le max de A et B
```

```
  R : Integer;
```

```
  -- resultat : le max de A et B
```

```
begin
```

```
  if A>B then R := A;
```

```
  else R := B;
```

```
  end if;
```

```
  return Renvoie(R);
```

```
end Max;
```

```
function Renvoie(A: in Integer) return Integer is
```

```
begin
```

```
  return A;
```

```
end Renvoie;
```

```
end maths;
```

```
  -- fin du package math
```

X.3.1 Paquetage maths (utilisation)

Exemple : Princ.adb – utilisation du package maths

-- Princ.ada : Essai du package maths

-- Auteur :

```

with maths, Ada.Text_IO, Ada.Integer_text_IO;  -- avec la package math
use Ada.Text_IO, Ada.Integer_text_IO;         -- on n'entre pas dans son domaine
procedure Principale is
  A, B : Integer;
begin
  Put("Entrer deux nombres : ");
  Get(A);
  Get(B);
  Skip_Line;
  Put("Le plus grand est : ");
  Put( maths.Max(A,B) );
  New_Line;
end Principale;

```

X.3.1 Paquetage maths (utilisation)

Exemple : Princ.adb – utilisation du package maths

-- Princ.ada : Essai du package maths

-- Auteur :

with maths, Ada.Text_IO, Ada.Integer_text_IO;

-- avec le package maths

use maths, Ada.Text_IO, Ada.Integer_text_IO;

-- on entre dans son domaine

procedure Principale **is**

A, B : Integer;

begin

Put("Entrer deux nombres : ");

Get(A);

Get(B);

Skip_Line;

Put("Le plus grand est : ");

Put(Max(A,B));

New_Line;

end Principale;

X.3.2 Éléments importants sur les paquetages

■ Unités de compilation et unités de bibliothèque

- **compilation séparée** (*separate compilation*)
- **unités de compilation** (*compilation units*)
- **bibliothèque** (*library*)
- **unités de bibliothèque**

- Toute spécification (ou sous-programme) mentionnée dans une clause de contexte doit être compilée avant l'unité mentionnant cette clause.
- Une spécification de paquetage doit être compilée avant le corps.

■ Conception d'un paquetage

- il doit fournir une solution d'un problème bien défini et, si possible, de faible complexité;
- il doit constituer une brique réutilisable, voire facilement extensible;
- la spécification doit être cohérente, former un tout;
- la spécification doit être simple, et faciliter la tâche de l'utilisateur du paquetage et non celle de son concepteur;
- le corps doit réaliser complètement et précisément la spécification.

X.3.3 Nombres rationnels (spécification)

Spécification cohérente du paquetage Nombres_Rationnels.

-- Ce paquetage permet le calcul avec les nombres rationnels

package Nombres_Rationnels **is**

type T_Rationnel **is**

-- Le type d'un nombre rationnel

record

Numerateur : Integer;

Denominateur : Positive;

-- Le signe est au numerateur

end record;

Zero : **constant** T_Rationnel := (0, 1);

-- Le nombre rationnel 0

-- Construction d'un nombre rationnel

function "/" (Numerateur : Integer; Denominateur : Positive) **return** T_Rationnel;

-- Addition de deux nombres rationnels

function "+" (X, Y : T_Rationnel) **return** T_Rationnel;

X.3.3 Nombres rationnels (spécification)

-- Soustraction de deux nombres rationnels

function "-" (X, Y : T_Rationnel) **return** T_Rationnel;

-- Multiplication de deux nombres rationnels

function "*" (X, Y : T_Rationnel) **return** T_Rationnel;

-- Division de deux nombres rationnels

function "/" (X, Y : T_Rationnel) **return** T_Rationnel;

-- Puissance d'un nombre rationnel

procedure Puissance (X:T_Rationnel; Exposant:Natural)
return T_Rationnel **is separate;**

-- Comparaisons de deux nombres rationnels

function "=" (X, Y : T_Rationnel) **return** Boolean;

function "<" (X, Y : T_Rationnel) **return** Boolean;

function "<=" (X, Y : T_Rationnel) **return** Boolean;

function ">" (X, Y : T_Rationnel) **return** Boolean;

function ">=" (X, Y : T_Rationnel) **return** Boolean;

end Nombres_Rationnels;

X.3.3 Nombres rationnels (corps)

-- Addition de deux nombres rationnels

function "+" (X, Y : T_Rationnel) **return** T_Rationnel **is**

 Le_P_P_M_C : Positive := P_P_M_C (X.Denominateur, Y.Denominateur);

begin -- "+"

return (X.Numerateur * (Le_P_P_M_C/X.Denominateur) +

 Y.Numerateur*(Le_P_P_M_C/Y.Denominateur), Le_P_P_M_C);

end "+";

-- Soustraction de deux nombres rationnels

function "-" (X, Y : T_Rationnel) **return** T_Rationnel **is**

 Le_P_P_M_C : Positive := P_P_M_C (X.Denominateur, Y.Denominateur);

begin -- "-"

return (X.Numerateur * (Le_P_P_M_C/X.Denominateur) -

 Y.Numerateur*(Le_P_P_M_C/Y.Denominateur), Le_P_P_M_C);

end "-";

X.3.3 Nombres rationnels (corps)

```

-- Pour la reduction en un nombre rationnel irreductible
function P_G_C_D ( X, Y : Positive ) return Positive is
    Diviseur_X : Positive := X;          -- Pour les soustractions
    Diviseur_Y : Positive := Y;
begin -- P_G_C_D
    while Diviseur_X /= Diviseur_Y loop
    -- PGCD trouve?
        if Diviseur_X > Diviseur_Y then
            Diviseur_X := Diviseur_X – Diviseur_Y;
        else
            Diviseur_Y := Diviseur_Y – Diviseur_X;
        end if;
    end loop;
    return Diviseur_X;          -- C'est le PGCD
end P_G_C_D;

```

X.3.3 Nombres rationnels (corps)

-- *Rendre un nombre rationnel irréductible après multiplication et division des deux nombres rationnels*

function Irreductible (X : T_Rationnel) **return** T_Rationnel **is**

 Le_P_G_C_D : Positive;

begin -- *Irreductible*

if X.Numerateur = 0 **then**

return (0, 1);

else

 Le_P_G_C_D := P_G_C_D (**abs** X.Numerateur, X.Denominateur);

return (X.Numerateur / Le_P_G_C_D, X.Denominateur/Le_P_G_C_D);

end if;

end Irreductible;

-- *Multiplication de deux nombres rationnels. Le resultat est un*

-- *nombre rationnel irréductible*

function "*" (X, Y : T_Rationnel) **return** T_Rationnel **is**

begin -- "*"

return Irreductible (X.Numerateur * Y.Numerateur, X.Denominateur * Y.Denominateur);

end "*";

X.3.3 Nombres rationnels (corps)

-- *Division de deux nombres rationnels. Le resultat est irreductible*

function "/" (X, Y : T_Rationnel) **return** T_Rationnel **is**

begin -- "/"

if Y.Numerateur > 0 **then** -- *Diviseur positif*

return Irreductible (X.Numerateur * Y.Denominateur, X.Denominateur * Y.Numerateur);

elsif Y.Numerateur < 0 **then** -- *Diviseur negatif, changer de signe*

return Irreductible (- X.Numerateur * Y.Denominateur, X.Denominateur * **abs** Y.Numerateur);

else -- *Division par zero!*

 ... -- *Lever une exception*

end if;

end "/";

-- *Comparaisons entre deux nombres rationnels: egalite*

function "=" (X, Y : T_Rationnel) **return** Boolean **is**

begin -- "="

return X.Numerateur * Y.Denominateur = X.Denominateur * Y.Numerateur;

end "=";

-- *Comparaisons entre deux nombres rationnels: inferieur*

function "<" (X, Y : T_Rationnel) **return** Boolean **is**

begin -- "<"

return X.Numerateur * Y.Denominateur < X.Denominateur * Y.Numerateur;

end "<";

X.3.3 Nombres rationnels (corps)

-- Comparaisons entre deux nombres rationnels: inferieur ou egal

function "<=" (X, Y : T_Rationnel) **return** Boolean **is**

begin -- "<="

return X.Numerateur * Y.Denominateur <= X.Denominateur * Y.Numerateur;

end "<=";

-- Comparaisons entre deux nombres rationnels: superieur

function ">" (X, Y : T_Rationnel) **return** Boolean **is**

begin -- ">"

return X.Numerateur * Y.Denominateur > X.Denominateur * Y.Numerateur;

end ">";

-- Comparaisons entre deux nombres rationnels: superieur ou egal

function ">=" (X, Y : T_Rationnel) **return** Boolean **is**

begin -- ">="

return X.Numerateur * Y.Denominateur >= X.Denominateur * Y.Numerateur;

end ">=";

end Nombres_Rationnels;

X.3.3 Nombres rationnels (corps)

-- *Puissance d'un nombre rationnel (Nombres_Rationnels-Puissance.adb)*

separate (Nombres_Rationnels) ou (nom de la procédure principale) si la fonction est dans le programme principale

procedure Puissance (X:T_Rationnel; Exposant:Natural) **return** T_Rationnel **is**

begin -- *Puissance*

return (X.Numerateur ** Exposant, X.Denominateur ** Exposant);

end Puissance;

Remarque : 1

on compile la spécification du paquetage

on compile le corps du paquetage

on compile programme principale (procedure principale)

on compile les procédures (separate)

on fait l'édition de liens

Puis, quelque soit ce que l'on modifie, on le recompile et on fait l'édition de lien.

Remarque 2 :

- Un fichier par procedure separate
- Nom du fichier Nombres_Rationnels-Puissance
- Dans le même répertoire que le programme principale

Remarque 3 :

Tout paquetage doit être accompagné de son mode d'emploi (règle d'utilisation des opérations exportées ainsi que les sources possibles d'erreurs générées lors de l'exécution de tout sous programme du paquetage

X.4 Paquetage enfant

```
package Nombres_Rationnels.Utilitaires is
```

```
    function "abs" ( X : T_Rationnel ) return T_Rationnel ;
```

```
end Nombres_Rationnels.Utilitaires;
```

```
package body Nombres_Rationnels.Utilitaires is
```

```
    function "abs" ( X : T_Rationnel ) return T_Rationnel is
```

```
    begin
```

```
        return ( abs X.Numerateur, X.Denominateur);
```

```
    end "abs";
```

```
end Nombres_Rationnels.Utilitaires ;
```

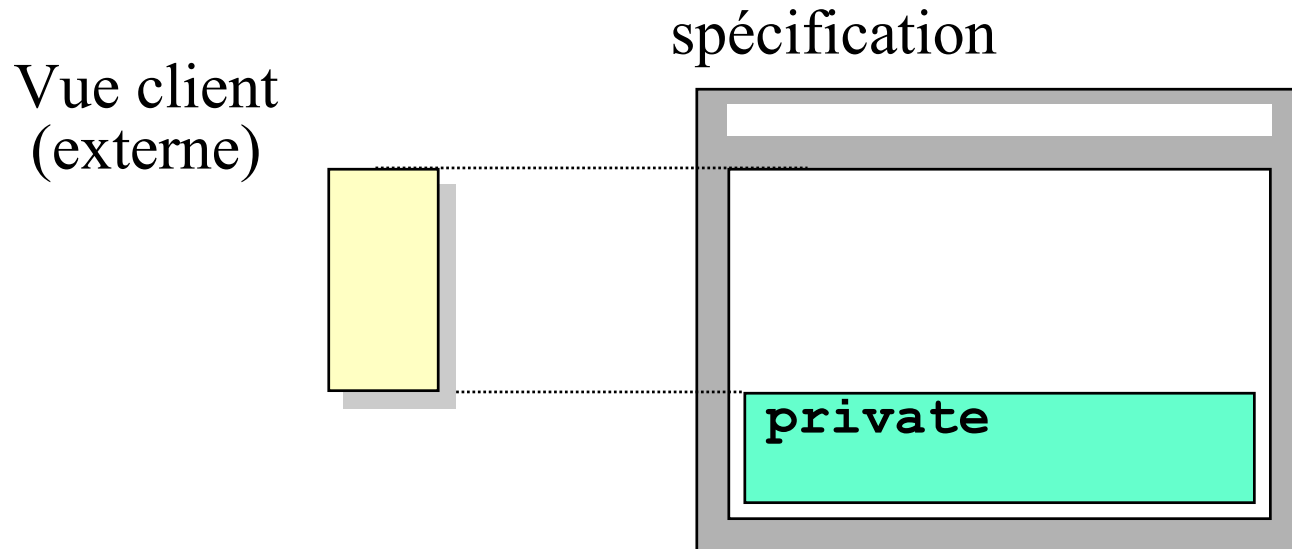
Remarque : 1

- Les éléments de la spécifications du paquetage parent sont visibles dans la spécification et le corps d'un paquetage enfant
- Les éléments du corps du paquetage parent ne sont jamais visibles pour le paquetage enfant
- Nom du fichier du paquetage enfant : Nombres_Rationnels-Utilitaires.adb

Remarque : 2

- la clause de contexte **with** d'un paquetage enfant comprend implicitement celle du paquetage parent, mais pas la clause **use**

- Partie privée
 - Abstraction, encapsulation.
 - Une partie de la spécification invisible pour le client.
 - Type de données abstrait.



Type limité privé

Seules les opérations de la partie visible sont accessibles.

Conséquence : pas d'affectation.

X.5.1 Paquetage nombres complexes (spécification)

package Nb_Complexe **is**

type T_Reel **is new Float**;

type T_Complexe **is private**;

I : **constant** T_Complexe;

function “+” (X,Y : T_Complexe) **return** T_Complexe;

private -- les détails du type sont cachés.

type T_Complexe **is**

record

Re : T_Reel := 0.0

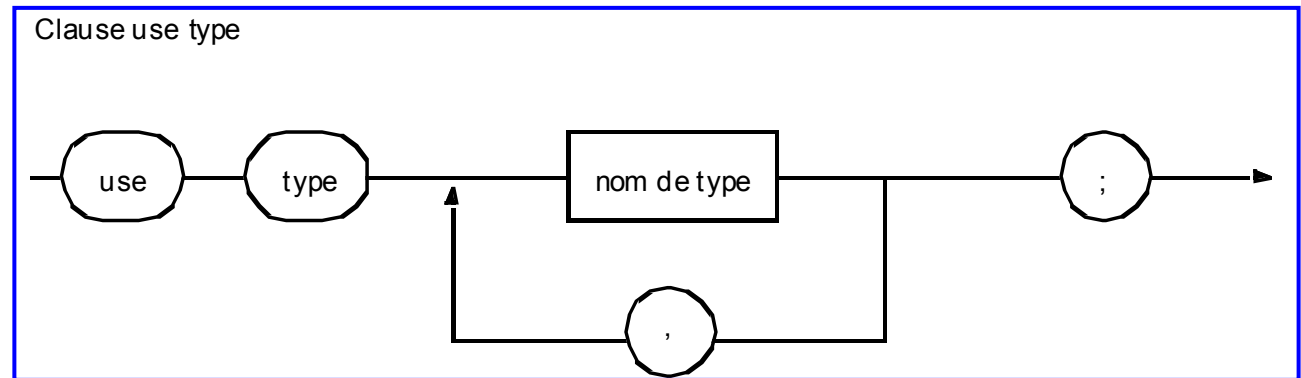
Im : T_Reel := 0.0;

end record;

I : **constant** T_Complexe := (0.0,1.0);

end Nb_Complexe;

X.5.2 Utilisation du paquetage Complexe avec clause use type



with NB_Complexe;

use type NB_Complexe.T_Complexe;

-- *La clause **use type** permet l'utilisation directe*

-- *sans préfixe, des fonctions opérateurs du type T_Complexe*

procedure Exemple **is**

Zero : **constant** Natural:=0;

Z1 : **constant** Complexe.T_Complexe := (1, 2);

Z2 : Complexe.T_Complexe ;

-- Le préfixe est cependant toujours possible pour T_Complexe

Z3 : Complexe.T_Complexe;

begin -- *Exemple*

-- *Affectations*

Z2 := Z1;

Z3:= Z1+Z2;

Sommaire

- Chapitre I : Présentation
- Chapitre II : Unités lexicales
- Chapitre III : Types et sous types
- Chapitre IV : Ordres (Sélection, cas, itération, ...)
- Chapitre V : Sous programmes
- Chapitre VI : Tableaux (array)
- Chapitre VII : Chaînes de caractères (String)
- Chapitre VIII : Articles (record)
- Chapitre IX : Pointeur
- Chapitre X : Fichiers (File)
- Chapitre XI : Paquetages simples (package)
- **Chapitre XII : Généricité**
- Chapitre XIII : Tâches

Paramétrage d'un S/P ou d'un paquetage à l'aide de

- types
- S/P
- objets
- valeurs

Développer un seul S/P ou paquetage dit générique.

L'instancier avec les paramètres désirés à chaque besoin.

→ le mot réservé **generic**

→ la liste des paramètres génériques formels

■ Instantiation

→ instruction **new**

→ tous les paramètres formels sont alors précisés

■ Les paramètres formels

```

type Item is private ;      -- indéfini
type Item is (<>) ;        -- discret
type Item is range <> ;    -- entier
type Item is digits <> ;   -- flottant
  
```

procedure Exemple **is**

```
Message : constant String (1 .. 9) := "Bonjour !";
```

generic

```
  with function Formater (S : String) return String;
```

```
  procedure Afficher (Message : in String);
```

```
  procedure Afficher (Message : in String) is
```

```
  begin -- Afficher
```

```
          Text_Io.Put (Formater (Message));
```

```
  end Afficher;
```

```
  function Hal_A_Dit (Mess : String) return String is
```

```
  begin -- Hal_a_dit
```

```
          return ("Hal a dit : < " & Mess & " >");
```

```
  end Hal_A_Dit;
```

```
  procedure Informer is new Afficher (Hal_A_Dit);
```

```
begin -- Exemple
```

```
  Informer (Message);
```

```
end Exemple;
```

Le concept de généricité permet de généraliser :

- les packages,
- les sous-programmes.

Exemple :

```
procedure Echange1 (a, b : in out Integer) is
    Temp : Integer;
begin
    Temp := a; a := b; b := Temp;
end Echange1;
```

```
procedure Echange2 (a, b : in out T_Mot) is
    Temp : T_Mot;
begin
    Temp := a; a := b; b := Temp;
end Echange2;
```

```
procedure Echange3 (a, b : in out Matrice) is
    Temp : Matrice;
begin
    Temp := a; a := b; b := Temp;
end Echange3;
```

```
procedure Echange4 (a, b : in out Float) is
    Temp : Float;
begin
    Temp := a; a := b; b := Temp;
end Echange4;
```

XI.1 Genericité

Il vaut mieux écrire une procédure d'échange générique :

```
generic
  type x is private;
procedure Echange(a, b : in out x) is
  Temp : x;
begin
  Temp := a; a := b; b := Temp;
end Echange;
```

Puis instancier avec le type désiré :

```
procedure Echange is new Echange (Integer);
procedure Echange is new Echange (mot);
procedure Echange is new Echange (matrice);
procedure Echange is new Echange (Float);
```

La surcharge de la procédure Echange est résolue grâce au type des paramètres d'appel :

```
i, j : Integer;      Echange (i, j); -- version Integer
f, g : Float;       Echange (f, g); -- version Float ...
```

Sommaire

- Chapitre I : Présentation
- Chapitre II : Unités lexicales
- Chapitre III : Types et sous types
- Chapitre IV : Ordres (Sélection, cas, itération, ...)
- Chapitre V : Sous programmes
- Chapitre VI : Tableaux (array)
- Chapitre VII : Chaînes de caractères (String)
- Chapitre VIII : Articles (record)
- Chapitre IX : Pointeur
- Chapitre X : Fichiers (File)
- Chapitre XI : Paquetages simples (package)
- Chapitre XII : Généricité
- **Chapitre XIII : Tâches**

Programme séquentiel : instructions exécutées dans l'ordre.

Parallélisme : plusieurs activités qui s'exécutent en parallèle.

Les tâches sont ces activités parallèles.

Les tâches :

- s'exécutent indépendamment
- peuvent se synchroniser (rendez-vous)

Vie d'une tâche

➤ Déclaration

☞ description de la spécification et du corps de la tâche

➤ Activation

☞ élaboration des déclarations (de la partie Déclaration) du corps de la tâche.

➤ Terminaison

☞ **Achevée** : le **end** final de la tâche est atteint.

☞ **Terminée** : toutes les tâches dépendantes sont terminées

Une tâche est une unité de programme.

```
-- Spécification
```

```
task Activité (signature) is
```

```
...
```

```
end Activité;
```

```
-- Corps
```

```
task body Activité (signature) is
```

```
...
```

```
end Activité;
```

La tâche dépend du cadre de déclaration.

Quitter le cadre : toutes les tâches dépendantes sont terminées.

Programme principal : tâche virtuellement appelée.

```
task type T_Activite (signature) is  
    ...  
end T_Activite;  
  
task body T_Activite (signature) is  
    ...  
end T_Activite;
```

Objet tâche

```
Tache_A, Tache_B : T_Activite;
```

Type accès et tâches

```
type Ref_Activite is access T_Activite;  
Tache_B : Ref_Activite := new T_Activite;
```

declare

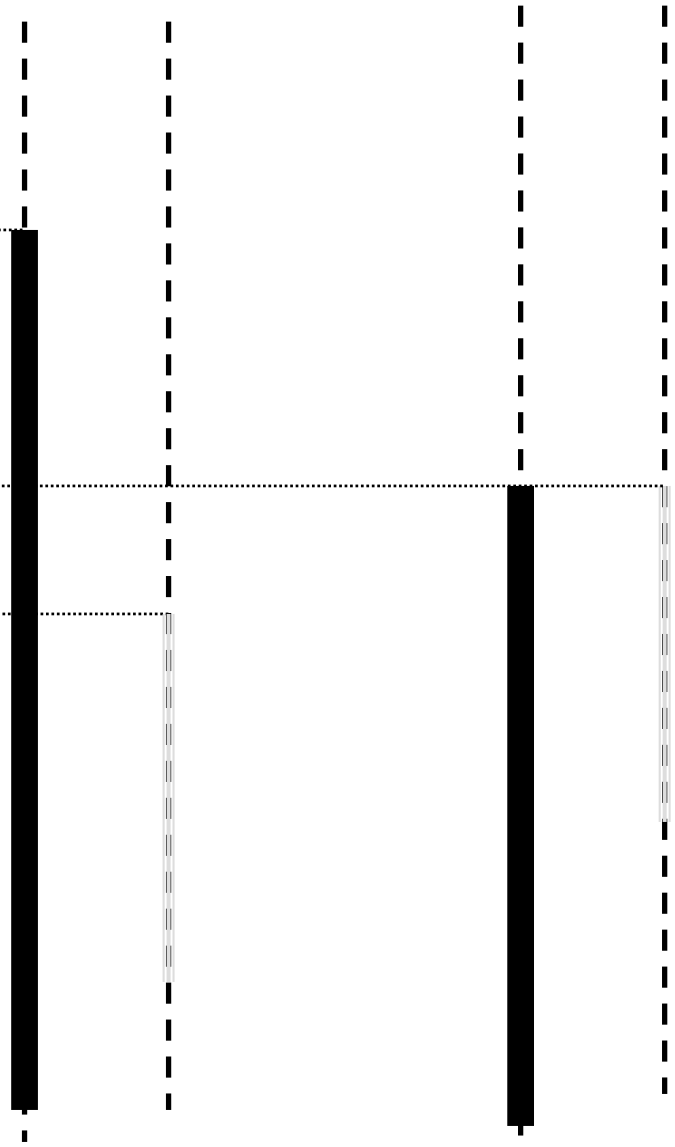
```
T_A : T_Activite;  
  
T_B : Ref_Activite :=  
      new T_Activite;
```

begin



T_A

T_B



Parent

declare

...

A :

T_tache;

B :

T_tache;

...

begin

...

end;

== activation

■ exécution

declare

begin

end

achevée

terminée

Tâche A

déclarée

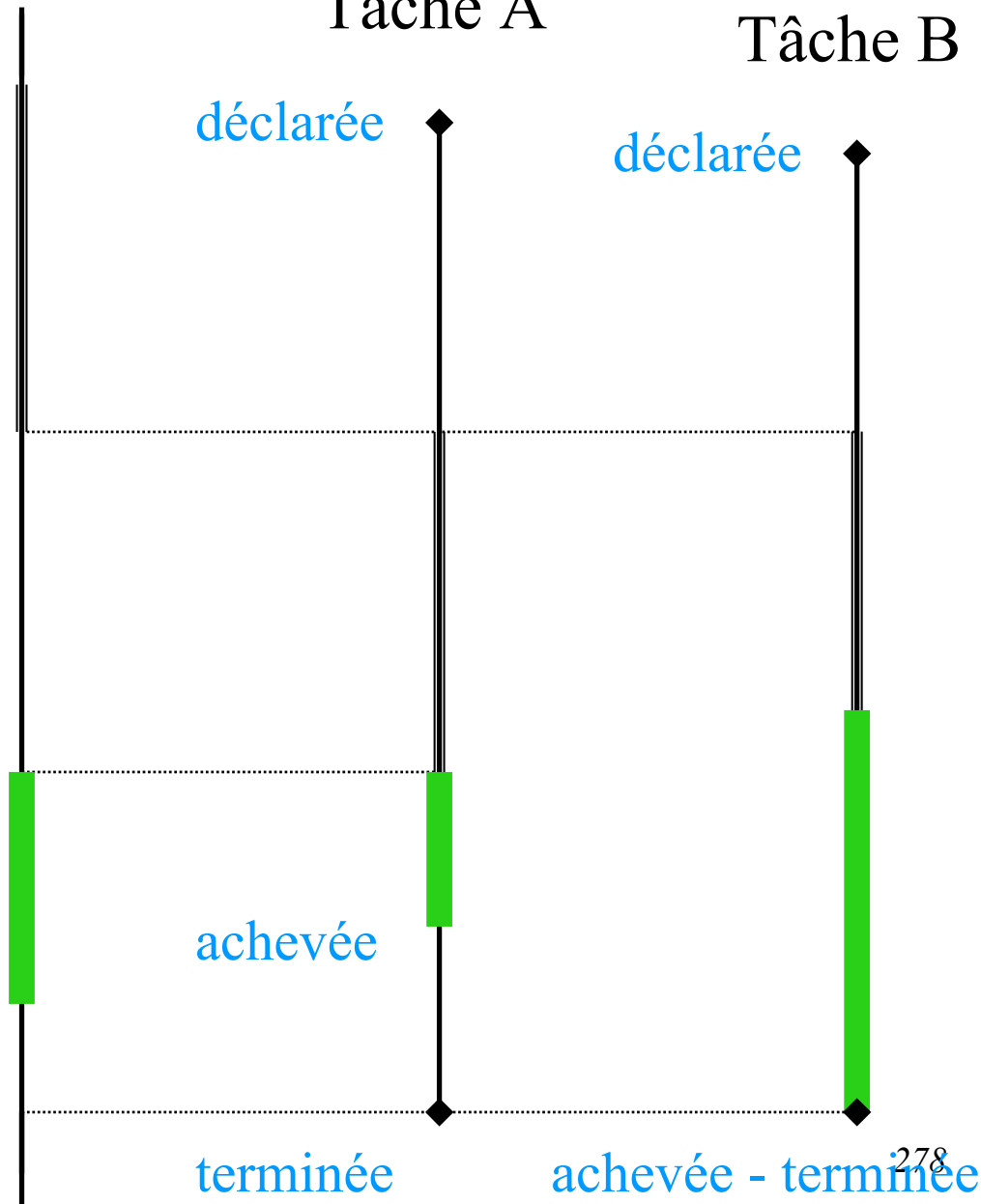
achevée

terminée

Tâche B

déclarée

achevée - terminée



Intéraction des tâches entre elles :

- **entry** : point d'entrée d'un Rdv (début du Rdv)
- **accept** : acceptation du Rdv, instructions à suivre .

Tâche A

```
B.Lecture;
```

Tâche B

```
task B is  
    entry Lecture (...);  
end B;  
  
accept Lecture (...) do  
    ... instructions  
end Lecture;
```

instruction **entry**

- déclarée dans la spécification de la tâche
- possède une signature
- s'appelle comme une procédure
- marque le début du Rdv

L'instruction **accept**

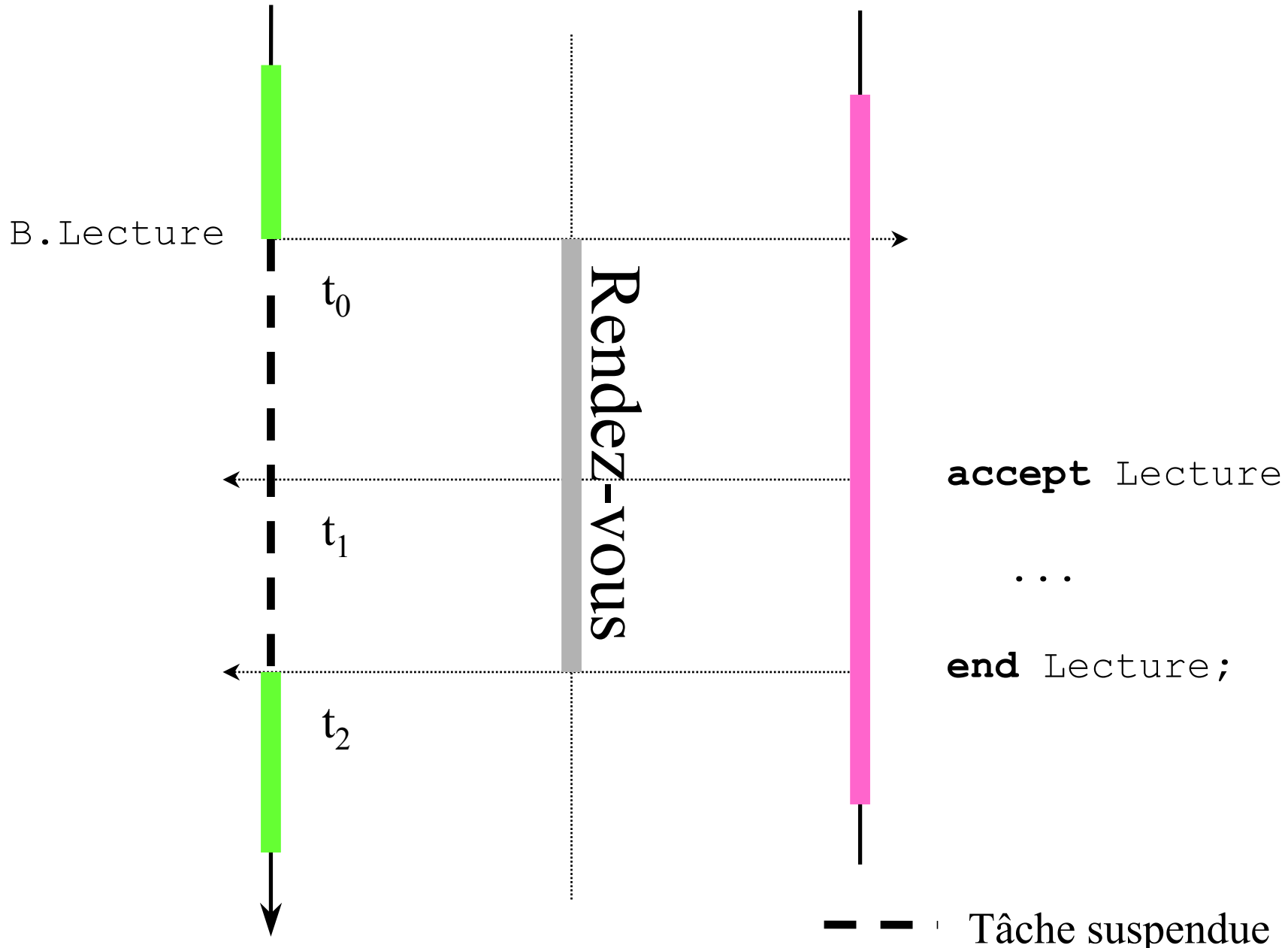
- contenue dans le corps de la tâche
- récupère la signature de l'entrée correspondante
- quand le **end** est atteint, le Rdv est fini

Appel de procédure : l'appelant (A) travaille.

Appel de tâche : l'appelé (B) travaille.

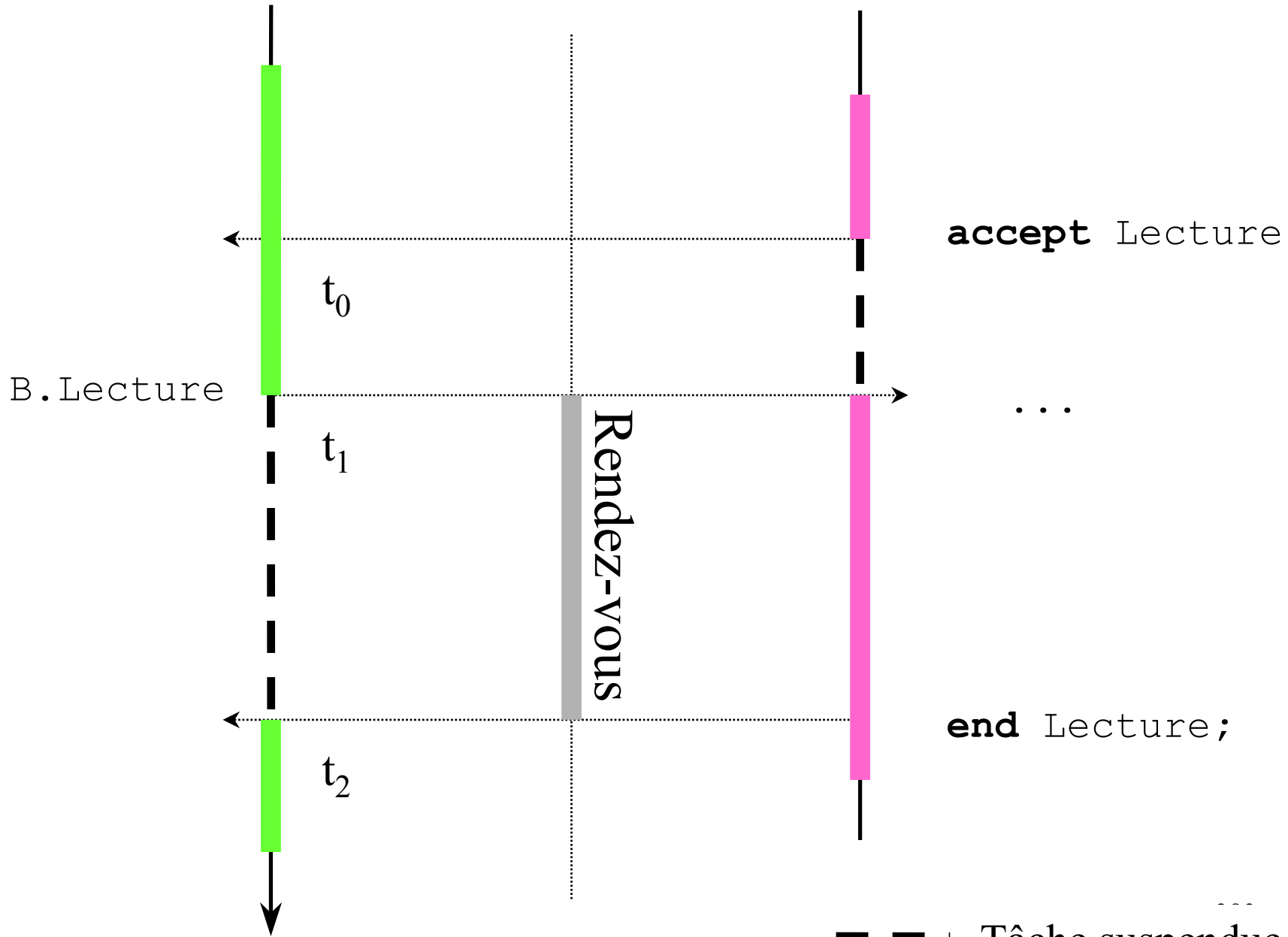
Tâche A

Tâche B



Tâche A

Tâche B



Paquetage Calendar

- ☞ Type Abstrait de Données
- ☞ type Time
- ☞ **exception** TIME_ERROR

Retenue d'une tâche

delay 3;

Suspension de la tâche pendant au moins 3 secondes.

- ☞ type prédéfini DURATION, type point-fixe
- ☞ intervalle minimum [-86400, 86400]
- ☞ DURATION's **small** < 20 ms (10**⁻⁹ GNAT 3.10)

Attente sélective

Choisir un Rdv parmi d'autres.

Instruction **select**.

```
select
    accept ...
end ;
...
or
    accept ...
end;
end select;
```

select

when Condition_1 => -- Garde

accept ...

or

accept ... -- *Garde toujours vraie*

or

when Condition_2 =>

accept ...

end select;

Evaluation des gardes sur l'instruction **select**.

PROGRAM_ERROR si toutes les gardes fausses

```
select
    accept ...
or
    accept...
or
    delay 5.0;
    ...
end select;
```

```
select
    accept...
or
    accept...
else
    ...
end select;
```

Equivalent à :

```
or
    delay 0.0
    ...
```

Appel d'entrée temporisé

```

select
    Ascenseur.Appel (Monter);
or
    delay 3*Minutes;
    Monter (Escalier);
end select;

```

● Appel d'entrée conditionnel

```

select
    Ascenseur.Appel (Monter);
else
    Monter (Escalier);
end select;

```

end final

terminate

abort

```
select
    accept ...
or
    terminate;
end select;
```

```
abort B.Lecture;
```

Terminaison normale

Terminaison anormale

- Standard (cf. [ARM A.1])
- Ada (cf. [ARM A.2])
- System et ses enfants (cf. [ARM 13.7])
- Interfaces et ses enfants (cf. [ARM B.2])
- Ada.Characters.Handling (cf. [ARM A.3.2])
- Ada.Strings (cf. [ARM A.4])
- Ada.Numerics et ses enfants (cf. [ARM A.5])
- Ada.Command_Line (cf. [ARM A.15])

- Le Language Reference Manual (LRM)
 - norme du langage Ada
 - référence précise au LRM dans les messages d'erreur à la compilation

- Livres
 - Programmation séquentielle avec ADA 95, [P. Breguet & L. Zaffalon 1999]
 - Cours enseigné chez Dassault Data Services [R. Chelouah 1999]
 - Vers ADA 95 par l'exemple [D. Fayard & M. Rousseau 1996]
 - Programmer en Ada, [J. Barnes, InterEditions 1988]
 - Introduction à ADA [C. Rapin 1988]
 - ADA Manuel de référence du langage de programmation [P.P.R 1987]
 - Introduction à ADA [P. Le Beux 1984]
 - Le langage ADA [D.J. David 1983]
 - Le langage ADA manuel d'évaluation [D. Le Verrand 1982]
 - Manuel de référence du langage ADA [Ministère Américain de la défense 1982]
 - ADA, Manuel complet du langage avec exemples [M. Thorin 1981]