Analyse et programmation langage ADA

Séminaire informatique 1ère année

R. Chelouah : rachid.chelouah@eisti.fr

Utilisation en mode diaporama
La page suivante donne le sommaire.
Pour accéder à un chapitre cliquer sur le lien correspondant
De n'importe quel transparent, on revient au sommaire en appuyant sur le logo EISTI

Date : 16/02/2006

Sommaire

Sommaire

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

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)
    - Entier (sans partie fractionnaire)
    - Réel (avec partie fractionnaire, éventuellement nulle)
  - Non décimal (bases 2 à 16 incluses)
    - Entier (sans partie fractionnaire)
    - Réel (avec partie fractionnaire, éventuellement nulle)

Nombres et leurs représentations en mémoire

Nombres entiers

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

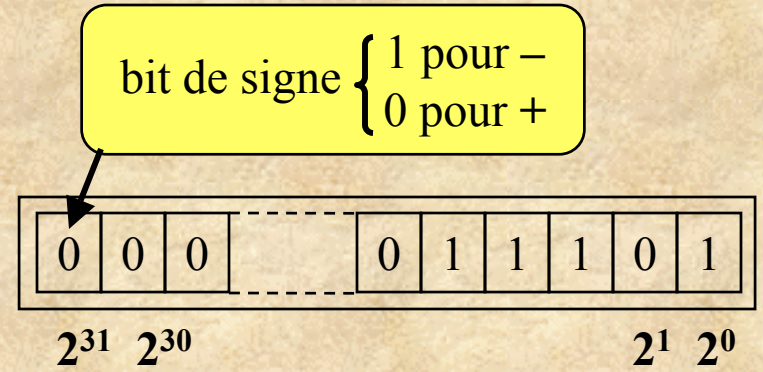
$$n = c_q 2^q + c_{q-1} 2^{q-1} + \ldots + c_1 2^1 + c_0 2^0$$

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

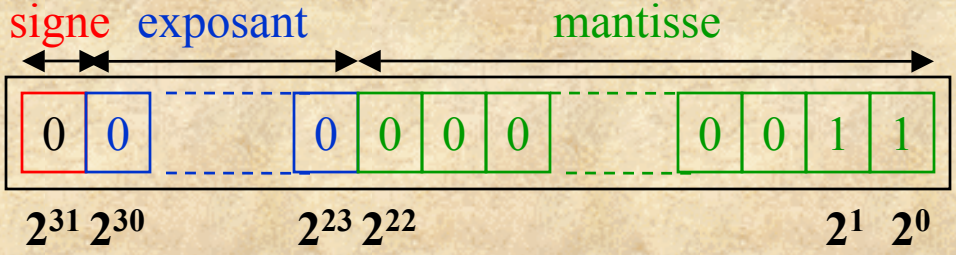
Le nombre 29 en binaire est stocké en mémoire de 32 bits



Nombres réels

Un nombre réel est représenté en machine sous forme : ± m E ±n

- d'une mantisse de 23 bits
- d'un exposant de 8 bits
- d'un bit de signe



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*

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 chassis
Construire le moteur ⟶ Construire la partie électrique
Construire la partie mécanique

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

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.

Spécificités du langage ADA

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

  valeur, Valeur et vaLeur représentent la même et unique variable

- **Les commentaires en ADA :**

  -- *Ceci est un commentaire qui*

  -- *s'étend sur plusieurs lignes*

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

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)

Apprentissage d'un langage

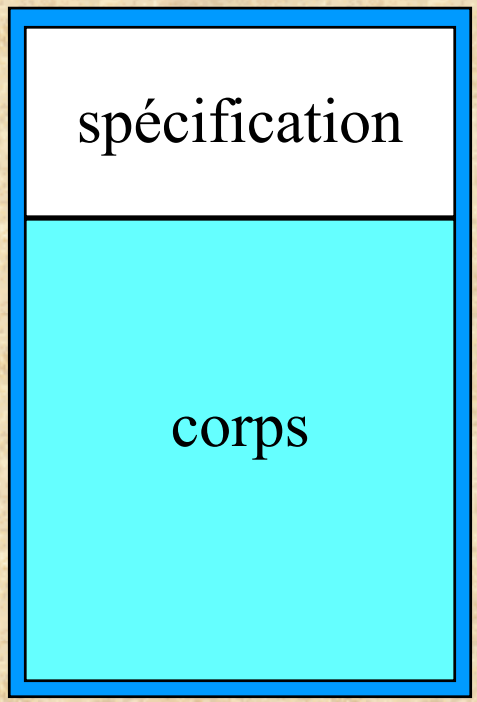
- 1.2.1 Éléments à traduire
- 1.2.2 Unités de programme
- 1.2.3 Types
- 1.2.4 Exemple
- 1.2.5 Dialogue programme utilisateur

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

Unités de programme

- Une unité de programme comporte deux parties :
  - la partie déclaration
    - ✓ spécification
    - ✓ élaboration des déclarations
  - la partie instructions
    - ✓ corps
    - ✓ exécuter les instructions



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 ;
```

Unités de programme

```
with Text_IO ; -- Appel aux bibliothèques

procedure afficher is

                    -- Partie Déclaration

begin --afficher
                    -- Partie Instructions

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

end afficher;
```

Unités de programme

```
-------------------------------------------------------------------------------
-- Fichier : afficher.adb
-- Application : Bonjour tout le monde!
-- 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 !");

end afficher;
```

Comment compiler et lancer un programme écrit en ADA Sous LINUX

- Editer
  - ✓ On ouvre un éditeur de texte vi ou emacs est on saisit le code de la page précédente
  - ✓ On sauvegarde ce fichier sous le nom de afficher.adb
- Compiler Passez dans le répertoire dans lequel vous venez d'enregistrer le fichier
  - ✓ on saisissez la commande suivante **gnatmake afficher.adb**
  - ✓ cette commande compile → édite les liens → construit votre code Ada
- Exécuter

Maintenant, si vous saisissez **./afficher**

vous obtenez le résultat suivant : Salut tout le monde !

Types

- Un **type** définit les valeurs que peut prendre un objet et les opérations qui lui sont applicables. Il existe 5 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

```
N : entier { définition }
B : booléen { déf. }
C : caractère { déf. }
R : réel  { définition }

CH : chaîne { déf. }
```

ADA

```
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

- **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
    -- 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" );
end Afficher_Hello;
```

commentaire de bloc

commentaire de ligne

Dialogue programme-utilisateur

- **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

Dialogue programme-utilisateur

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.

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...
```

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 ;
```

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

Résultat
Votre moyenne est de : 3.9

- 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 ; Ada.Float_Text_IO.Get(Note_Info) ;	4.5 3 --*action de vider le buffer d'entrée* --*sinon lecture de la valeur 3 dans Note_Info* 4.1

Résultat
Votre moyenne est de : 4.34

Dialogue programme-utilisateur

- 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;
```

Dialogue programme-utilisateur

- 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;
-- 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); Skip_Line;
                Put ( "Donnez la note d'informatique : " );
                Get (Note_Info); Skip_Line;
```

Clauses de contexte

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

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;
```

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

Mise en page d'un programme

- En tête d'un programme

```
-------------------------------------------------------------------------------------------------
-- Nom fichier       : Calcul_Gain.adb
-- Auteur            : Dupond Jean
-- Date              : 26/10/2008
-- 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
-------------------------------------------------------------------------------------------------
```

Mise en page d'un programme

- 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 depasser
-- 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

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
```

Mise en page d'un programme

- 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, …).

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 changer 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;
```

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 :

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

```
-- 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;
```

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);
     Skip_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
- Chapitre IV : Ordres(Sélection, cas, itération, ...)
- Chapitre V : Sous programmes

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

Exemple d'unités lexicales

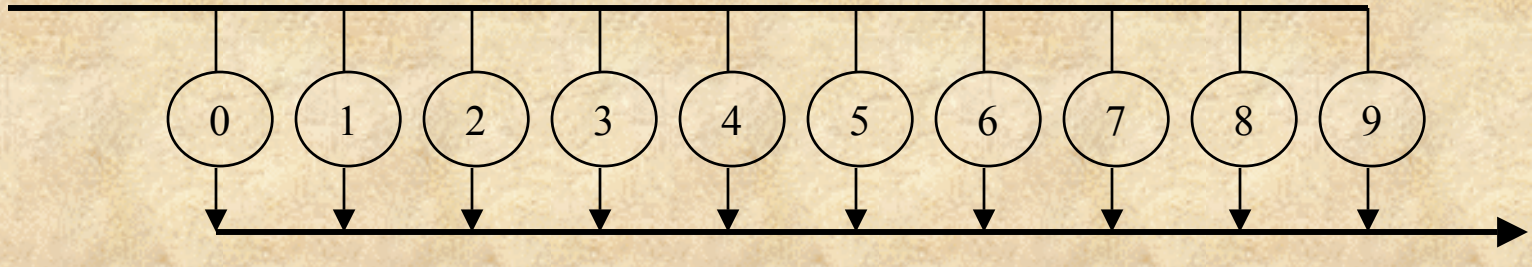
Une *unité lexicale* (mot d'un langage de programmation), est le plus petit élément d'un programme, non décomposable.

- 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 chaines de caractères : "blabla", "1","", " "
- Délimiteurs : +, -, *, :=, >=

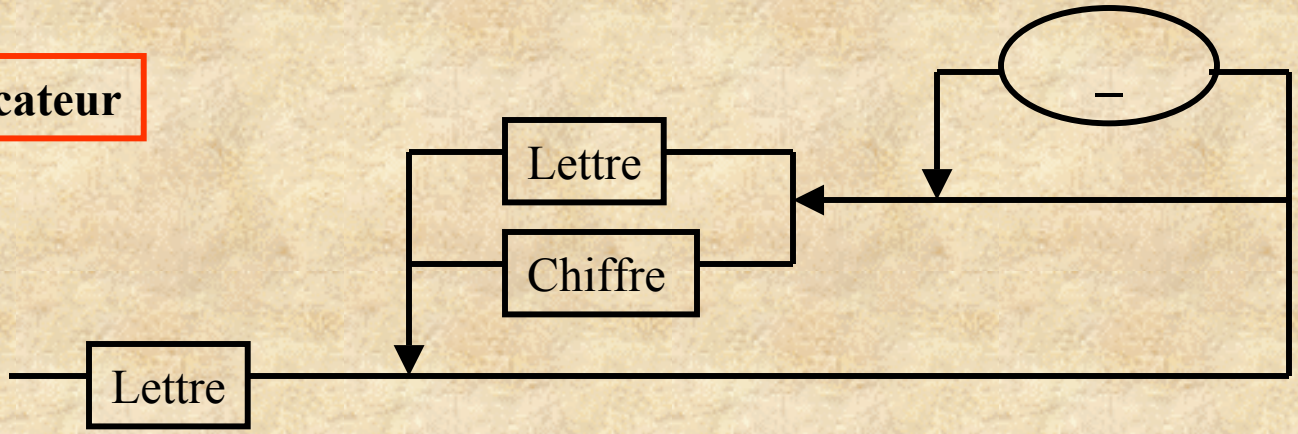
Identificateurs (*ou id.*), mot choisie

- 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				

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 seuleument 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

Introduction

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.

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 **rem** (-B) = A **rem** B et (-A) **rem** B = -(A **rem** B).
    - Le mot résevé **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 +.

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 **rem** 2 et 4 **mod** 2 sont deux expression de valeur 0
  - 5 **rem** 2 et 5 **mod** 2 sont deux expression de valeur 1
  - 5 **rem** (-2) est une expression de valeur 1
  - (-5) **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 .

Type entier (opération)

- 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;
```

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;
```

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*

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)
  - ✓ Sort_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

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*

Utilisation d'un type énumération

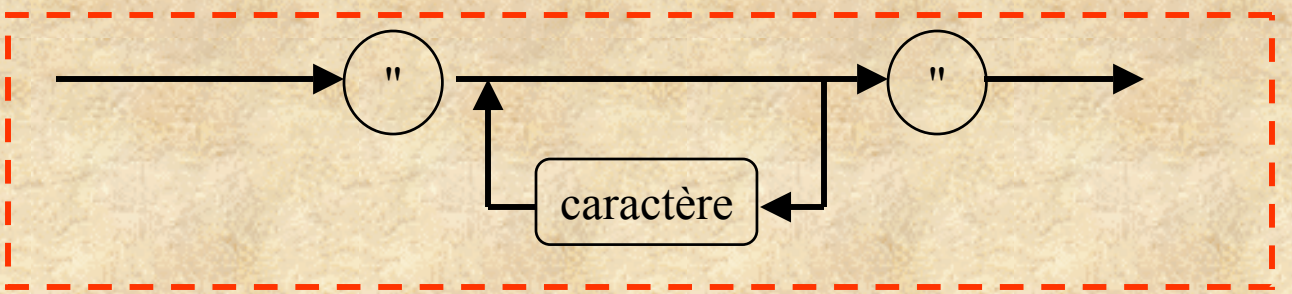
```
procedure Exemple is
   type T_Fruits is (Orange, Banane, Pomme);
   type T_Legumes is (Carotte, Poireau);
   Un_Fruit  : T_Fruits ;
   Un_Legume : T_Legumes ;

    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;
```

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

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

Charactères (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'VAl(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 < charactères ordonnés

Lecture et Ecriture

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');
```

Présentation du 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

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

ADA

- **opérateurs sur les booléens :**
  - **not (priorité 1)**
  - **and (priorité 2)**
  - **or (priorité 3)**

  ADA évalue toujours la totalité de l'expression
- **comparaisons (priorité 4) :**
  - = /=
  - < >
  - <= >=

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 seulse 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.3.3.4 Traduction des opérations élémentaires

Notation algorithmique

- **opérations sur les entiers :**
  + - * / div mod ^

exemple d'expression :
A^8 + B * C div D + E mod 3

- **opérations sur les réels + - * /**

ADA

- **opérateurs sur les entiers :**
  + - * / / mod **

A8+((B*C) / D) + (E mod 3)**

- **opérateurs sur les réels + - * /**

Conversions de types

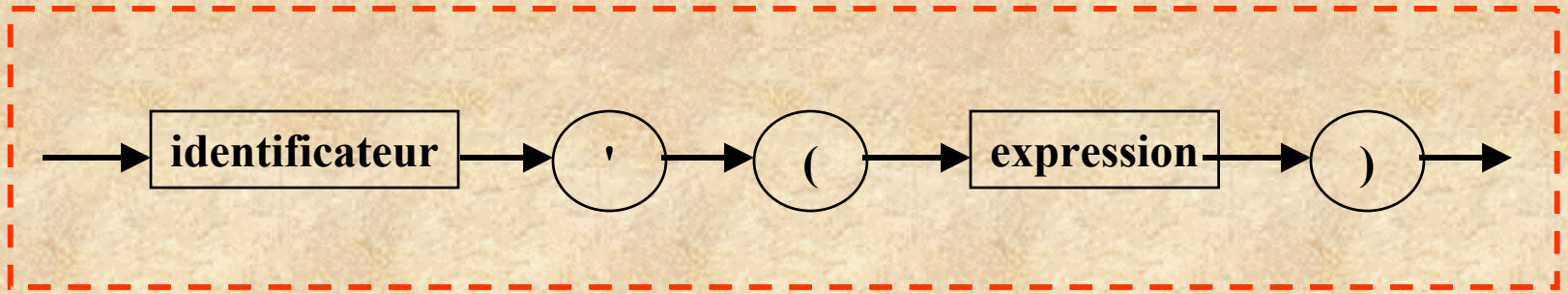
Pour faire une conversion donnée => nom_de_type(expression)

- 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

```
...
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.
...
```

Expressions

- **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;

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 réelprédéfini*

  with Ada.Float_Text_IO; **use** Ada. Float _Text_IO;

Sommaire

- Chapitre I : Présentation
- Chapitre II : Unités lexicales
- Chapitre III : Types
- **Chapitre IV : Ordres(Sélection, cas, itération, ...)**
- Chapitre V : Sous programmes

Chapitre IV : **Ordres**

Un *ordre* est un constituant de programme qui entraine *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

Affectation

Une affectation qui permet de mettre une valeur donnée ou la valeur calculée d'une expression donnée dans une région de la mémoire centrale accessible par le nom de la variable.

- Une affectation : variable = expression

Notation algorithmique

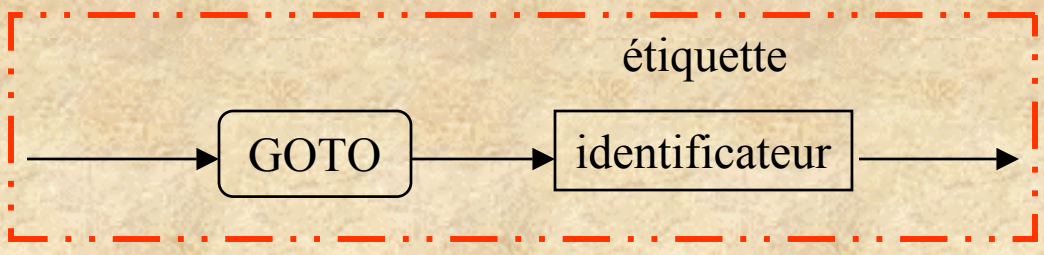
```
Programme Test
        Variables A, B : Entier
Début
        A ← 5
        B ← 2
        A ← B
        B ← 2A + 1
Fin
```

ADA

```
procedure  Test
    A, B : Integer ;
begin
        A = 5;
        B = 2;
        A = B;
        B = 2*A +1;
end
```

Branchement

- Le *branchement* est défini ainsi :



Exemple

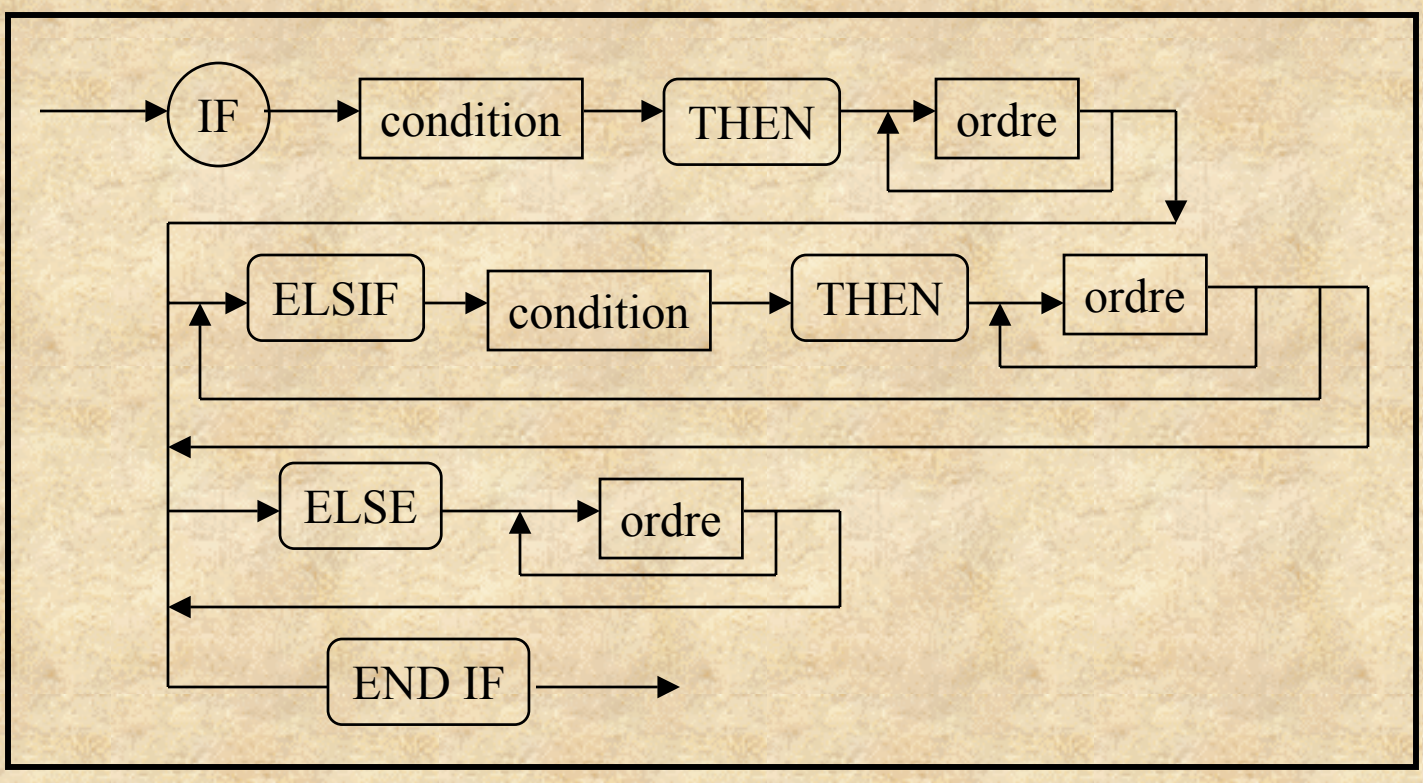
```
begin
    if Y = 0 then
        GOTO  Label;
    end if;
    D := X/Y ;
    << Label >>   D := Max ; --étiquette figurant ailleurs (lourde et laide)
end ;
```

- 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.

Alternative

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 ← Y
sinon MAX ← 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 ← Y
  X = Y : M ← Y+X
  X > Y : M ← 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 ← Y
  (X = Z) et (Z > Y):
            M ← Y+X
  autre cas :
            M ← X-Y
fin selon
```

ADA

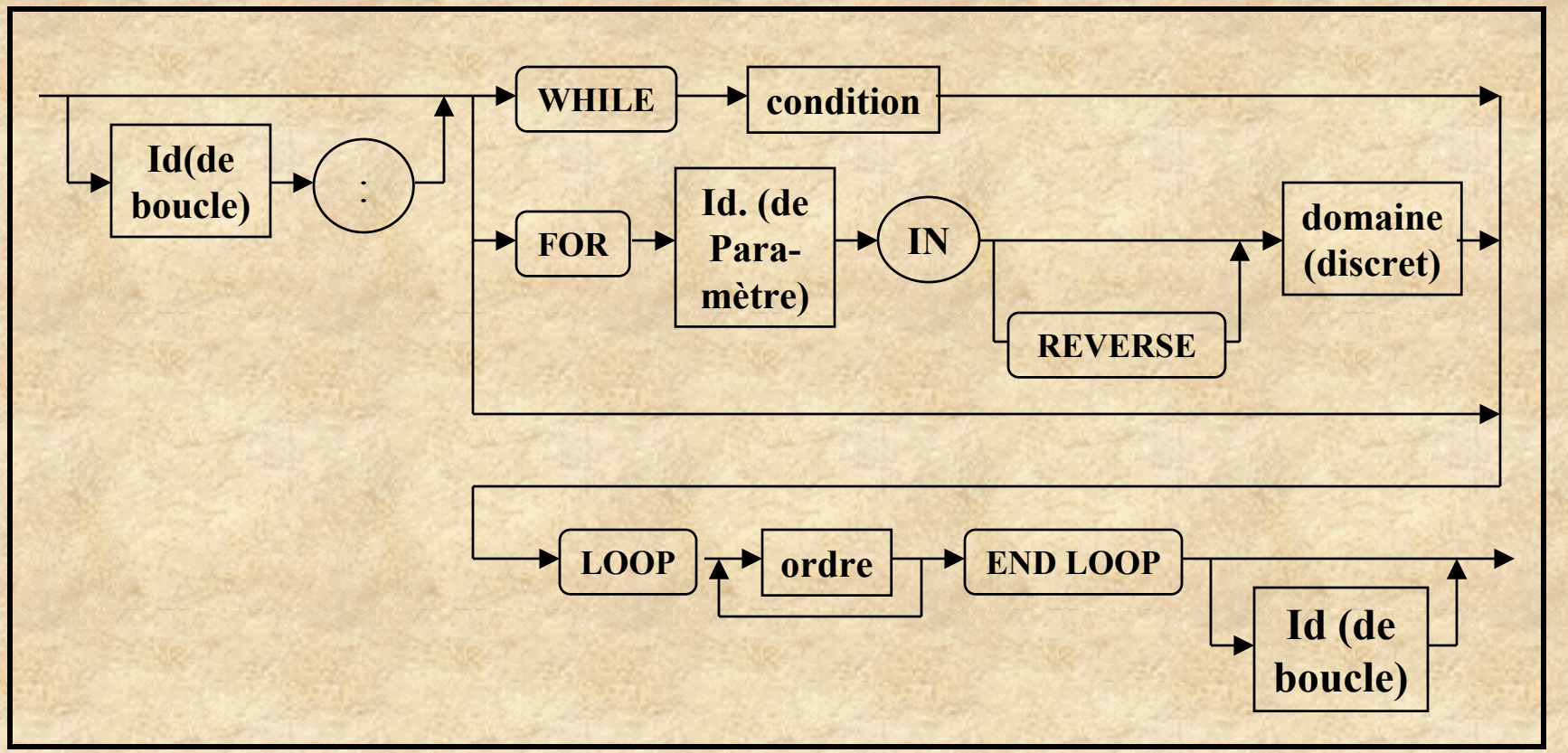
```
-- selon  X, Y
if  X < Y then
   M := Y;
else if  X=Y and Z>Y then
    M := Y+X;
else
   M := X-Y;
end if
```

Boucle

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

Boucle

- 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 condition vérifiée faire
                action
        FinTantque
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

Pour Compteur ← Initial **à** Final **Pas** Valeur
    action
FinPour

Pour Compteur I allant de 4 à 1 (4 fois)
    AV(50+I)
    GA(90+2*I)
FinPour

ADA

```
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'expression_2 – expression_1 + 1
- La variable de boucle (variable d contrôle) est déclarée implicitement, du type des borne 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' expression_1 a une valeur supérieure à expression_2, la boucle n'est pas effectuée
- Si la valeur de expression_1 ou expression_2 est modifiée par une itération, le nombre d'itérations ne change pas!

La boucle générale loop

Notation algorithmique

répéter
    action
Jusquàceque bouléen

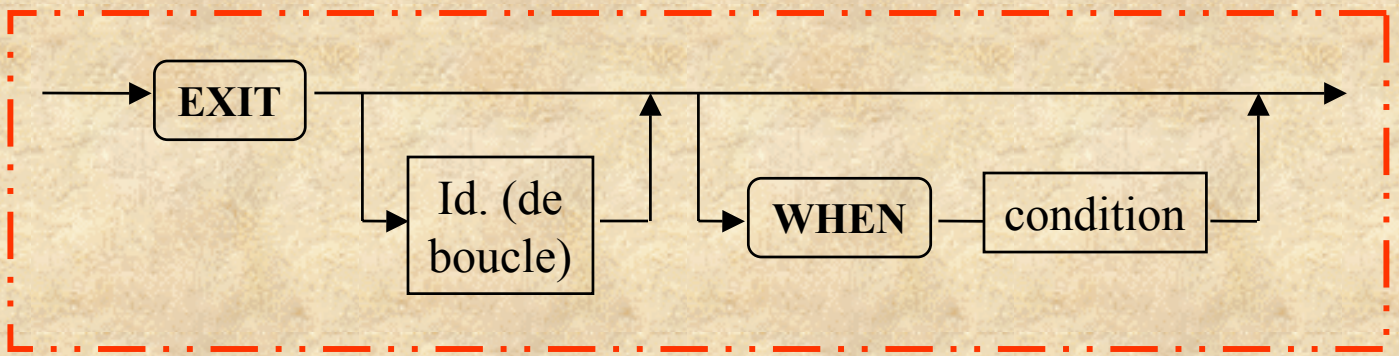
```
N ← 10
i ← 1
S ← 0
répéter
    S ← S + i
    i ← i + 1
jusquàceque i > N
```

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;
```

Sortie de boucle

- 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.

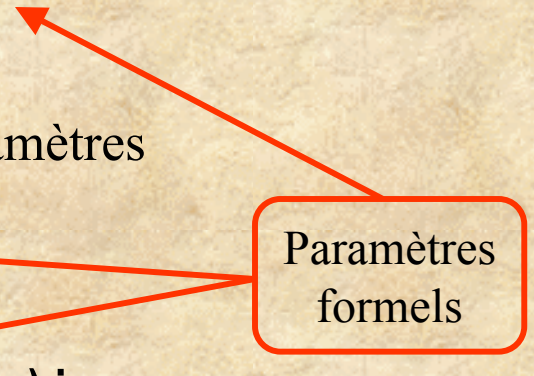
Sommaire

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

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) ;



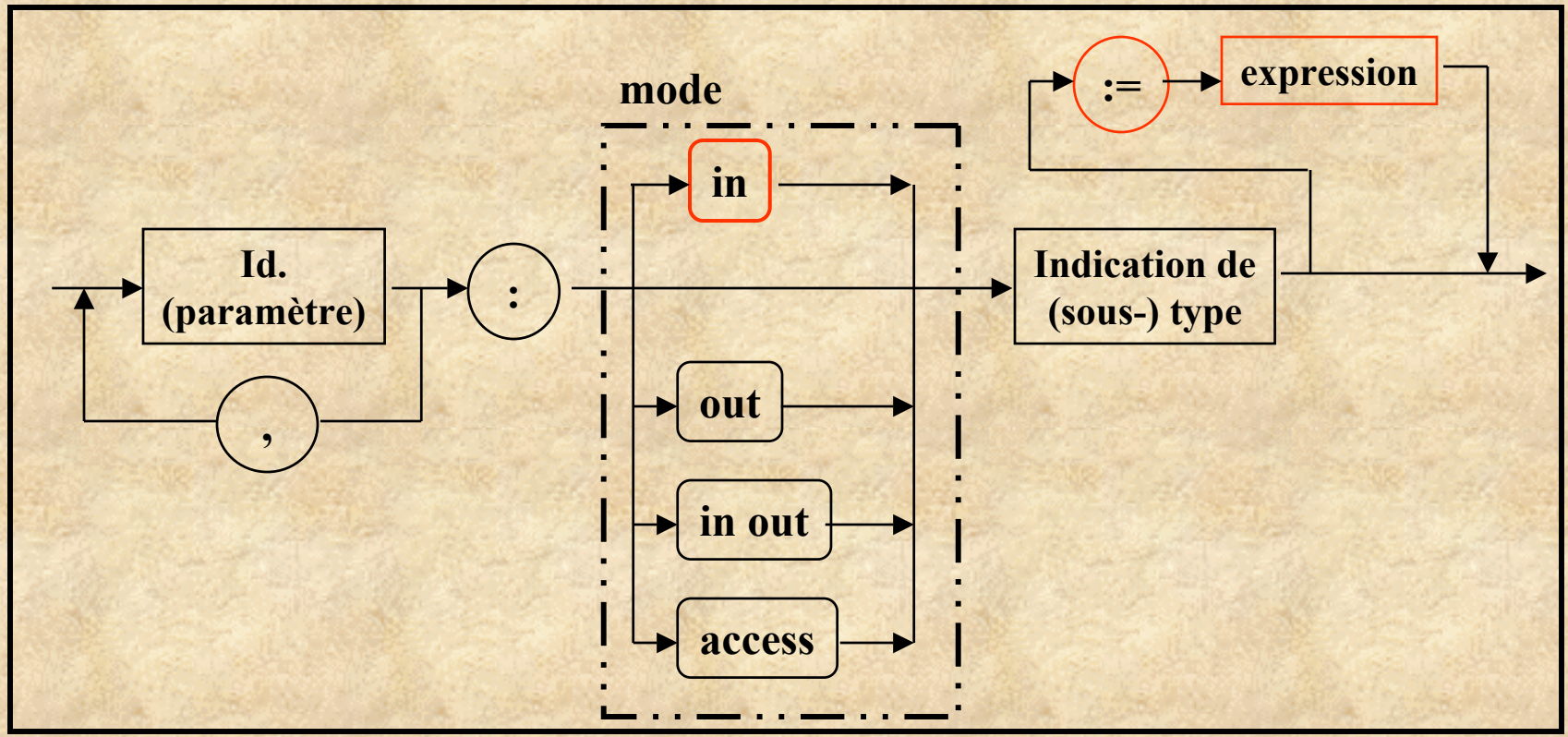
Exemple :

```
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	X : **in** typeparam
Elaboré Y : typeparam	Y : **out** typeparam
Modifié Z : 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

Traduction des paramètres formels

- 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

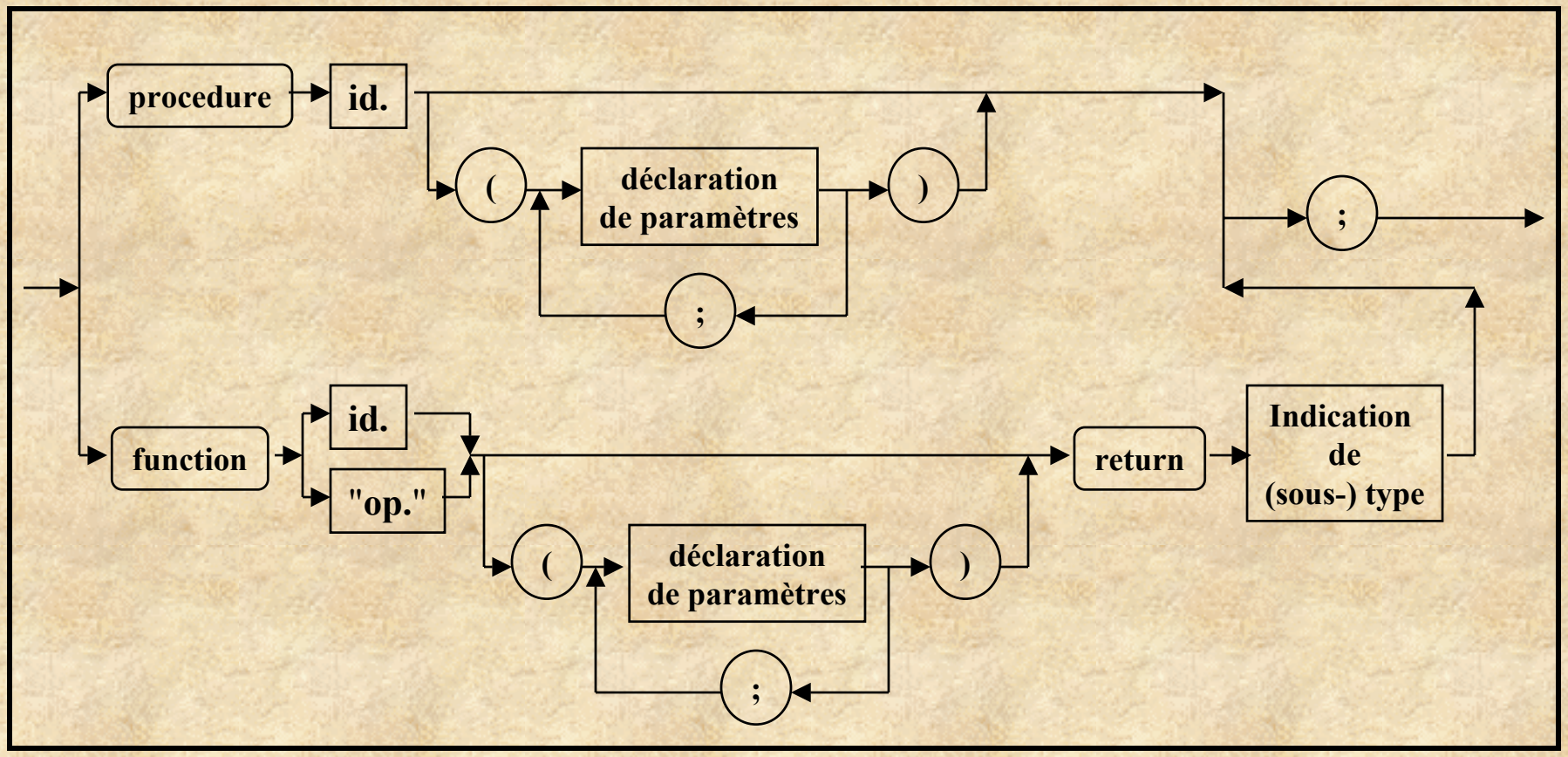
```
X : in Float := 1.2;
Y : in  Float := 1.5;
Z : out  Float   -- ( valeur de retour)
```

Convention :

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

Déclaration de sous programme

- 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, lorqu'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**.

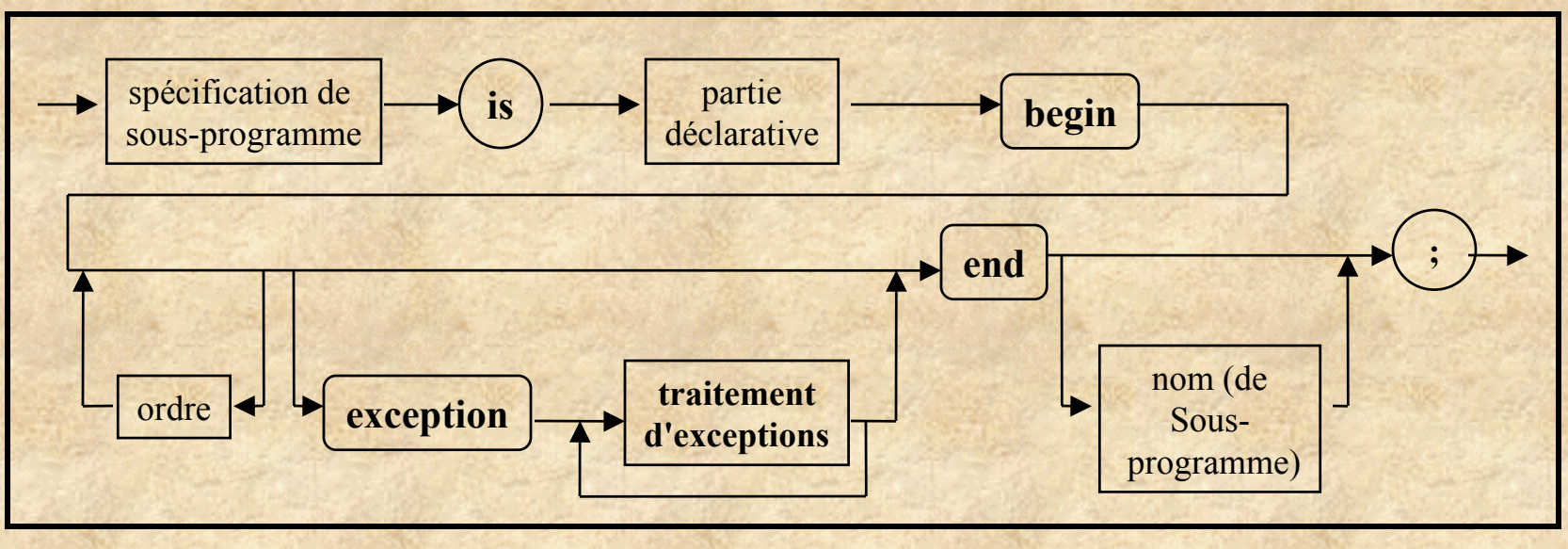
function Somme(X, Y : **in** TDecimal) **return** TDecimal ;

- Une *procedure* est un sous programme qui, lorqu'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.

Fonctions

- 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**.

Procédures

- Déclaration

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

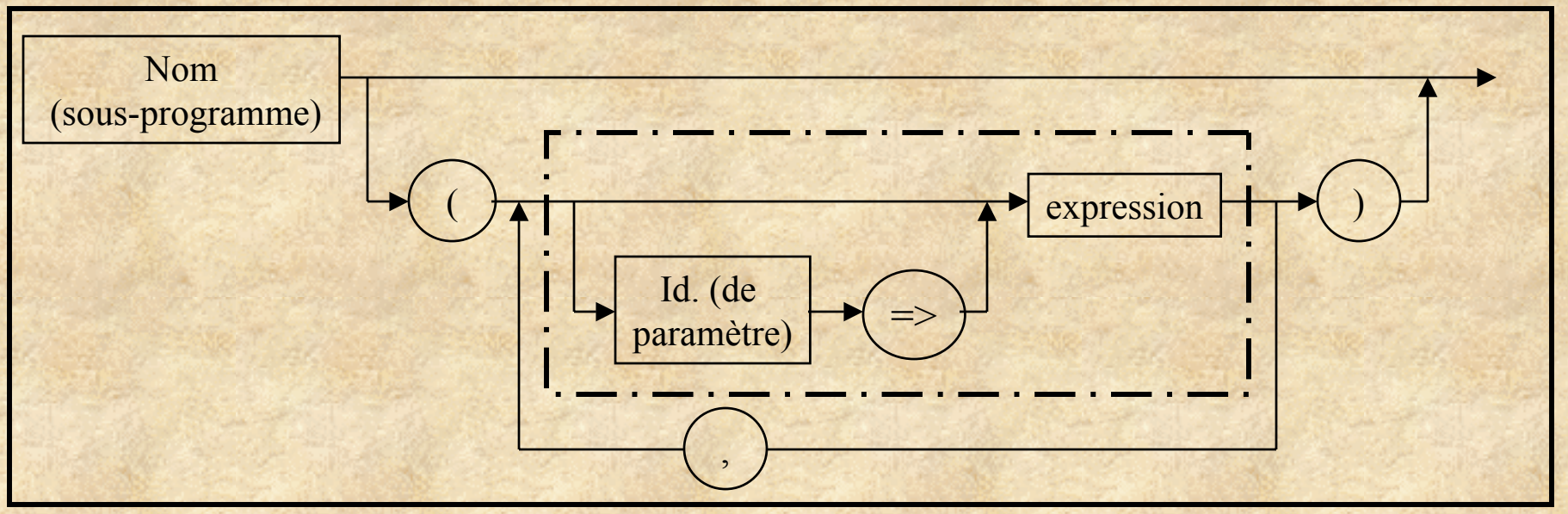
- Corps

```
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 donner 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);                    New_Line(Spacing =>3);          Skip_Line;
Z := Sin(X+Y);             X := Random(Generator =>Generateur);
```

Résultat d'appel de sous programme

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 :

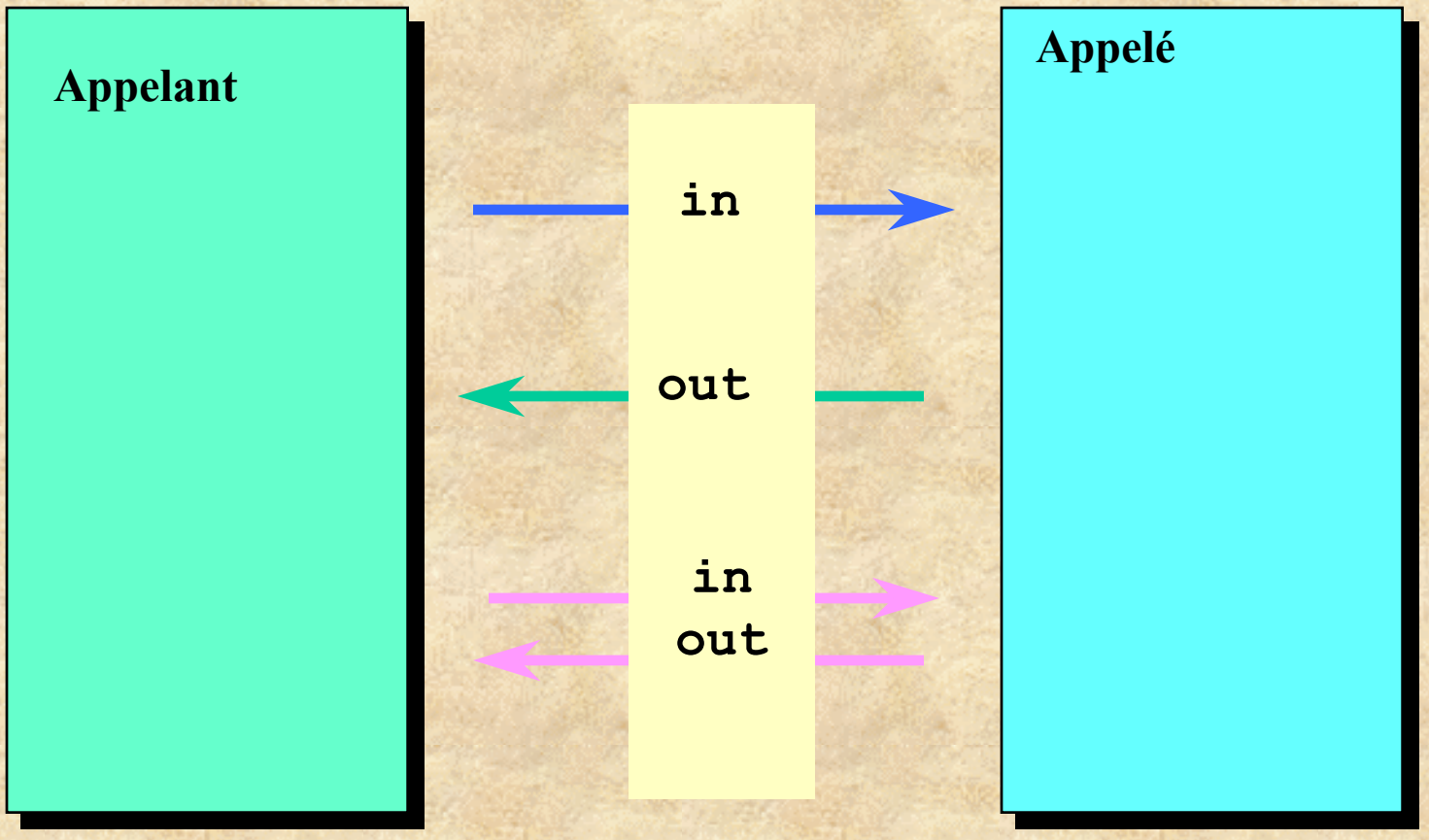
  Cube (J);

- 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 <==> INTEGER à l'entrée du sous-programme et conversion inverse au retour.

Appel de sous programme



Signature ≡ contrat entre l'appelant et l'appelé

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 remplacez ce code par une procédure `Permuter`