Séance 2_bis

Introduction au langage Ada

Objectifs :

- ✓ Éléments Syntaxiques du langage.
- ✓ Construction d'un programme.
- ✓ Comprendre la structure générale d'un programme Ada, ainsi que le mécanisme des unités de compilation.
- ✓ Connaître un sous-ensemble d'Ada, suffisant pour écrire des programmes simples.
- ✓ Explorer les instructions de base.

Ada 2005

- Programmation Orientée Objet
- Types access
- Temps-réel, sécurité et criticité
- Usage général
- Environnement prédéfini et interfaçage

Programmation Orientée Objet

- Préserve les forces d'Ada pour la construction de systèmes sécuritaires
  - Distinction entre les types spécifiques et à échelle de classe
  - Aiguillage statique par défaut, aiguillage dynamique seulement lorsque nécessaire
  - Frontières fortes autour des modules
- Amélioration du modèle objet
  - Redondance cyclique
  - Hiérarchie pour l'héritage multiple
  - Programmation Orientée Objet et concurrence
  - Notation préfixée
  - Surchage accicentelle

Programmation Orientée Objet : Redondance cyclique

- Impossibilité de déclarer des types cycliques parmi plusieurs paquetages
  - Existant en Ada 83
  - Plus proéminent avec l'introduction des unités enfants et les types étiquetés
  - Solution de rechange encombrantes

Programmation Orientée Objet : Redondance cyclique

- Solution de rechange encombrantes

```
package Married is type Wife_Type;
   type Wife_Access is access all Wife_Type'Class;
   type Husband_Type;
   type Husband_Access is access all Husband_Type'Class;

   type Husband_Type is tagged
        record
                Wife : Wife_Access;
        end record;

   type Wife_Type is tagged
        record Husband : Husband_Access;
        end record;
end Married;
```

Programmation Orientée Objet : limited with

- Accorde la visibilité limitée du paquetage
  - Les types se comportent comme des types incomplets
  - Restrictions sur l'usage la vue limitée : pas de use, pas de renommage, seulement pour la spécificaiton, etc.)
  - Cycles permis entre les clauses limited with

- Autres changements : types étiquettés incomplets
  - Peut être utilisé comme paramètre
  - Toujours passé en référence

Programmation Orientée Objet : limited with

```
limited with Wife_Pkg;
package Husband_Pkg is
   type Husband_Type is tagged
     record
        Wife : access Wife_Pkg.Wife_Type;
     end record;
end Husband_Pkg;
```

```
limited with Husband_Pkg;
package Wife_Pkg is
   type Wife_Type is tagged
      record
         Husband : access Husband_Pkg.Husband_Type;
      end record;
end Wife_Pkg;
```

Héritage multiple : hiérarchie de types

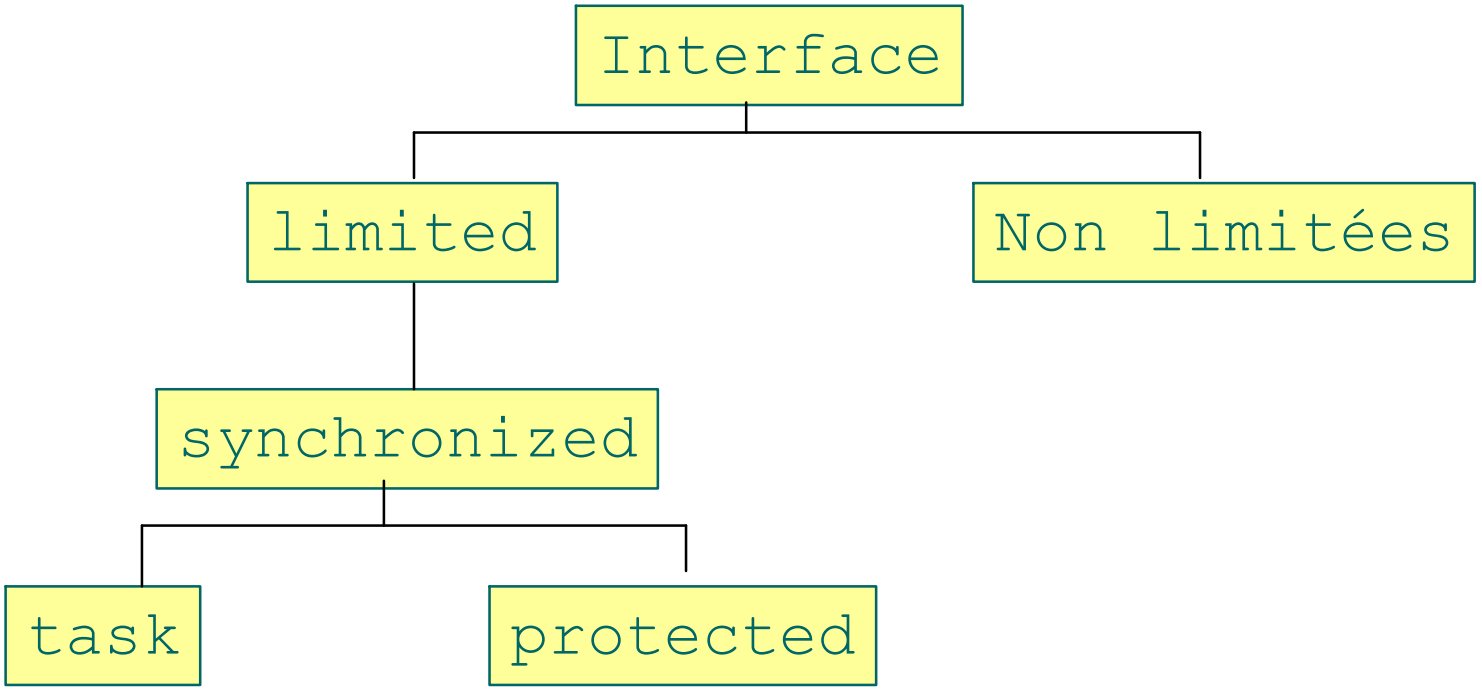
- Héritage multiple lourd en Ada 95
- Java et C# possèdent un mécanisme d'héritage multiple léger : interface
  - Sans coût sur le temps d'exécution
  - Pas de conflit pour le code hérité de plusieurs parents
- Ajout d'interface, (ressemble aux types abstraits mais avec héritage multiple)
  - Deuxième parent lors de la dérivation
  - Type à échelle de classe
  - Support pour la composition d'interface
  - Pas de composant, ni d'objet
- Autres changements : procédure null
  - Une procédure null ne doit pas obligatoirement être surchargée

Héritage multiple : hiérarchie de types

```
package Q is
   type Int is interface;
   procedure Op ( X : int) is abstract;
   procedure N ( X : int) is null;
   procedure Action ( X : Int'Class); -- opération à échelle de
classe
end Q;
```

```
package P is
   type T is tagged
      record
        ….
      end record;
   procedure Op ( X : T) is abstract;

end P;
package PQ is
   type TI is new P.T and Q.Int with private;

   procedure Op ( x : TI);
private
  type TI …;

end PQ;
```

Héritage multiple : généralisation avec la concurrence



Programmation orientéé objet : notation objet.Méthode

- Notation préfixée

```
package P is
      type T is tagged  private;
       procedure Méthode ( Obj : T);
  private
      type T is tagged
         record
            comp : integer;
         end record;
  end P;
```

```
with P;    -- pas de use
  procedure Main_oo is
     Objet : P.T;
  begin
     Objet.Méthode;    -- équivalent à Méthode(Objet);
end Main_oo;
```

Programmation orientéé objet : surcharge et remplacement

- Surcharge accidentelle (overloading & overriding)

```
package P is
      type Grand_Parent is abstract tagged null record;
      type Classwide_Ptr is access all Grand_Parent'class;
      procedure Operation1(This : access Grand_Parent) is abstract;
      procedure Operation2(This : access Grand_Parent);

      type Parent is new Grand_Parent with null record;
      overriding procedure Operation1(This : access Parent);
      -- overriding is optional, indicates that this procedure must
      -- override an inherited method not overriding
      procedure Operation3(This : access Parent);
      -- not overriding is optional, indicates that this procedure
must NOT
      -- override an inherited method
      type Kid1 is new Parent with null record;
      overriding
      procedure Operation1(This : access Kid1);
      type Kid2 is new Parent with null record;
      overriding
      procedure Operation2(This : access Kid2);
   end P;
```

Types access

- La plupart des langages permettent la conversion d'une référence d'une sous-classe à la référence de la super-classe
  - Ada nécessite la conversion explicite (dégrade la lisibilité)
- Permettre des types anonymes dans de nouveaux contextes
  - Composante d'article et de tableau
  - Renommage d'objet
  - Résultat des fonctions
- Réduire les conversions explicites
- Éviter la prolifération des types accès

Types access

- Réduire les conversions explicites

```
type Parent is tagged null record;
type Parent_Access is access all Parent'Class;
type Child is new Parent with null record;
type Child_Access is access all Child'Class;
P : Parent_Access;
C : Child_Access;
```

et ensuite

```
P := C; -- illegal!!
P := Parent_Access(C); -- legal
```

Types access

- Réduire les conversions explicites

```
type Parent is tagged null record;
type Parent_Access is access all Parent'Class;
type Child is new Parent with null record;
type Child_Access is access all Child'Class;
P : access Parent'Class;
C : access Child'Class;
```

et ensuite

```
P := C; -- légal
```

- Composant d'article

```
type List_Node is
   record
      …
      Next : access List_Node;
   end record;
```

Types access not null et constant

- Pointeur anonyme en pouvant avoir la valeur null et accès constant

```
type Int_Access is access Integer;
type Book_Access is not null access Book;
type Book_Const_Access is not null access constant Book;
procedure Catch(Ball : not null access Throwable);
function Color(Car : not null access constant Automobile) return
Color_Type;
-- Color ne peut changer la valeur de car.all
Y : not null Int_Access := new Integer'(5);
Z : not null Int_Access; -- illégal !
B : Book_Const_Access := new Book'(…);

B.Price := 99.99;  -- est illégal
```

Types access anonyme à des sous-programmes

- Accès anonyme à des sous-programmes

```
function Integrate (
      Fn : access function (X : Float) return Float;
      Lo, Hi : Float) return Float;
```

- Valeur non nulle à des pointeurs

```
type Non_Null_Ptr is not null access T;
procedure Pass_By_Ref (Y : not null access constant
  Rec);
-- Any pointer to a graph may be passed to the display
-- routine, including null.
procedure Display (W : access Window;
                   G : access constant Graph'Class);

function Get (W : access Window) return not null access
  Person;
```

Ajout d'usage général (Unions C)

- Unchecked Union : article avec un discriminant ne faisant pas partie de l'article (interface les unions de C)
- pragma Unchecked_Union évite que le discriminant soit stocké comme composant de l'article

```
union {
    spvalue double;
    struct {
        length int;
        first *double;
    } mpvalue;
} number;
```

```
type Number (Kind : Precision) is
record
   case Kind is
     when Single_Precision =>
         SPValue : Long_Float;
     when Multiple_Precision =>
         MP_Value_Length : Integer;
         MP_Value_First :
   Access_Long_Float;
    end case;
end record;
pragma Unchecked_Union (Number);
```

Ajout d'usage général : opérations vectorielles et matricielles

- ISO/IEC 13813 définit les vecteurs et matrices réel et complexes pour Ada 83
  - Pas de support de base pour l'algèbre linéaire de base
  - Pas fournit par les fournisseurs de compilateurs
- Intègre cette fonctionnalité à l'Annexe G (Numerics)
  - 2 nouvelles unités prédéfinies:

    Ada.Numerics.Generic_Real_Arrays et

    Ada.Numerics.Generic_Complex_Arrays
  - Adapté à Ada 05
  - Support pour l'algèbre linéaire de base : inversion, résolution, valeurs propres
  - Peut-être ajouté en interface à un paquetage existant ou implanté du tout au tout

Ajout d'usage général : bibliothèques de conteneurs

- Un langage avec des conteneurs définis dans le langage améliore la portabilité et l'utilisabilité du langage
- Fonctionnalité de base
  - Tri
  - Vectors
  - Hashed maps
  - Ordered sets
  - Doubly-linked lists
  - Variantes pour les types définis et indéfinis
- Techniques d'implantation non spécifiées
  - Limites de performance autorisées
  - Insister sur la sécurité
- Composantes additionnelles peuvent être définies par un International Workshop Agreement

Ajout d'usage général : fonctionnalités du système d'exploitation

- Faciliter l'utilisation des fonctionnalités des systèmes d'exploitation modernes
- Nouveau paquetage prédéfini : Ada.Directory_Operations
  - Donne accès à l'arborescence du système de fichier
  - Créer, éliminer, copier, et renommer des fichiers ou des répertoires
  - Décomposer et composer les chemins des fichiers et des répertoires
  - Vérifier l'existence, modifier la taille des fichiers et l'étampe temporelle
  - Itérer sur les fichiers et répertoires
- Nouveau paquetage Ada.Environment_Variables
  - Lire et écrire les variables d'environnement
  - Itérer sur les variables d'environnement

Ajout d'usage général : exception

- Ajout du pragma Assert
  - Exception Assertion_Error

```
pragma Assert ( L.Next /= null, «La liste contient plus de 2
   éléments »);
```

- Spécification d'un message avec l'instruction raise

```
raise Une_Exception with, «Le message d'erreur approprié»);
```

Ajout d'usage général : petits changements

- Ajout au paquetage Ada.Real_Time
  - Fonctions `Minutes` et `Seconds` pour les tests de minuteries
- Ajout d'une fonction dans `Ada.Text_IO` pour lire une chaîne de caractères sans se préoccuper de la longueur

```
declare
   Line : String := Ada.Text_IO.Get_Line;
begin
```

- Nouveau paquetage Ada.Strings.Unbounded.Text_IO;

Éléments lexicaux

Objectifs :

- Comprendre la structure générale d'un programme Ada, ainsi que le mécanisme des unités de compilation.
- Connaître un sous-ensemble d'Ada, suffisant pour écrire des programmes simples.
- Explorer les instructions de base.

ÉLÉMENTS LEXICAUX

- Un élément lexical, ou jeton, est la plus petite unité syntaxique d'un programme.
- Les éléments lexicaux d'Ada sont :
  - les identificateurs,
  - les mots réservés,
  - les littéraux,
  - les délimiteurs.

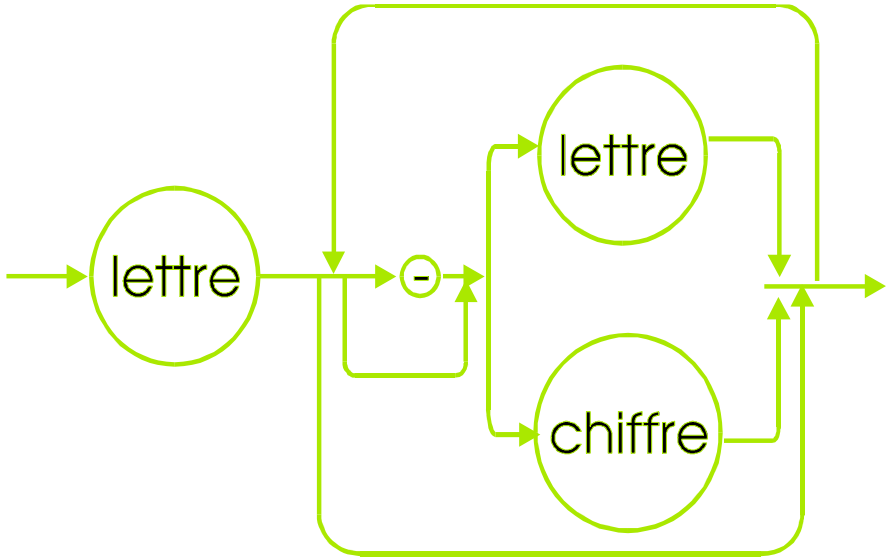
IDENTIFICATEURS -- NOMS

- Les noms identifient des objets, variables, types, sous-programmes, et attributs.
- Un nom est composé d'un identificateur.
- Un identificateur commence par une lettre, suivi d'une suite de lettres, chiffres ou "_".
- Lettre (majuscule ou minuscule)
- La syntaxe d'un identificateur en BNF est:

  identificateur ::= lettre { [_] (lettre | chiffre)}

UTILISATION DE BNF's

- Les BNF's peuvent être représentées par un diagramme:



- Notation des BNF
  - {a} 0, ou plusieurs a
  - a | b 'a' ou 'b'
  - [a] 'a' optionnel

FORMAT -- PROGRAMME Ada

- La syntaxe est libre (format libre)

- Les mots réservés ne peuvent pas être utilisés à d'autres fins.

MOTS RÉSERVÉS

- 63 mots réservés (+5 avec Ada 95 +3 avec Ada 2005)
- Pour limiter le nombre de mots réservés, Ada utilise certains mots réservés dans plusieurs sens, (e.g. ***null***) ou des suites de mots réservés (e.g. ***or else***)
- Les types prédéfinis, e.g. *INTEGER*, ne sont pas des mots réservés.

DÉLIMITEUR & SÉPARATEURS

- Les délimiteurs sont des caractères spéciaux, tandis que les séparateurs sont des "blancs".
- Les délimiteurs sont : ' ( ) * + , - . / : ; < = > |
- De plus, les longs délimiteurs sont:

  => .. ** := /= >= <= >> << <>

- **Les séparateurs sont** : un ou plusieurs espaces, ou les caractères de tabulation ou de fin de ligne.

COMMENTAIRES & PRAGMA

- Un commentaire commence par "--" et se termine à la fin de la ligne.

- Un ***pragma*** transmet des informations au compilateur,

  ex:

  pragma *Page;*
  pragma *Preelaborate;*
  pragma *List(OFF);*
  pragma *Import (Fortran, Sin);*
  pragma *Suppress (Range_Check, On=>Indice);*
  pragma *Optimise (Time);*
  pragma *Inline (convertir, sous_programme);*

PRAGMA

- Le développeur peut ajouter librement des pragmas à ceux déjà définis
  - ***pragma*** *SOURCE_INFO (ON);*
- Les pragmas peuvent apparaître n'importe où une déclaration ou un énoncé peut apparaître.

LITTÉRAUX NUMÉRIQUES

- 2 classes de littéraux numériques:

  littéraux réels

  littéraux entiers

  littéral_numérique ::= littéral_décimal | littéral_basé

  littéral_décimal ::= entier [. entier]

  [(E | e) (+ | - |) entier]

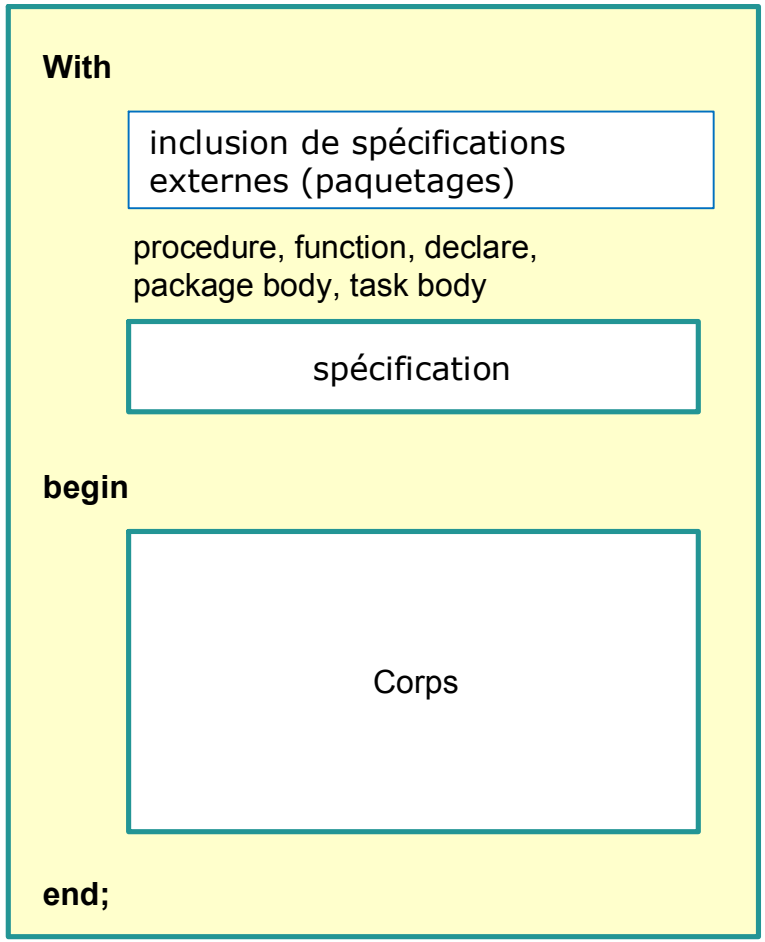
- Un littéral réel contient un point (.).

STRUCTURE D'UN PROGRAMME Ada

- Dans plusieurs langages, un programme est constitué d'un programme, et d'un certain nombre de sous-programmes.
- Une telle organisation est inefficace pour de gros systèmes et les logiciels modulaires.
- Ada permet de grouper les déclarations et les sous-programmes dans des **unités séparées**, et de les compiler séparément.
- Une unité peut importer des déclarations et peut les exporter vers d'autres unités.
- Un programme principal est un sous-programme qui n'est pas importé.

STRUCTURE D'UN PROGRAMME (Suite)

```
with Ada.Text_IO;

procedure Bonjour is
begin
   Ada.Text_IO.Put_Line ("Bonjour le monde");
   Ada.Text_IO.Put_Line ("Comment allez-vous");
end Bonjour;
```



STRUCTURE D'UN PROGRAMME (Suite)

```
with Ada.Text_IO;
use Ada.Text_IO;
procedure Bonjour is
begin
   Ada.Text_IO.Put_Line ("Bonjour le monde");
                Put_Line (" Comment allez-vous");
end Bonjour;
```

Exemple

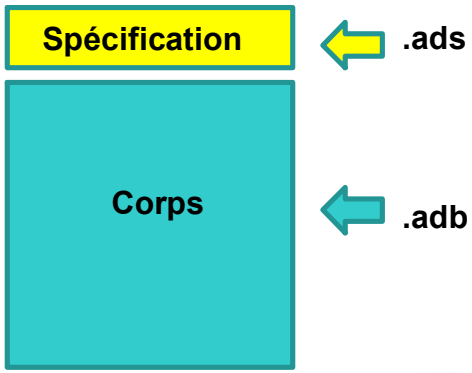
```
-------------------------------------------------------------------------------
-- Fichier : Bonjour.adb
-- Application : Bonjour le monde
-- Auteur : Noureddine Kerzazi
-- Liste des packages utilisés par notre programme
-------------------------------------------------------------------------------
with Text_IO;
procedure Bonjour is
begin -- Afficher
    Text_IO.Put_Line(" Bonjour le monde !");
exception -- traitement des exceptions
when others => Text_IO.Put_Line ("Erreur dans le programme principal");

Bonjour ;
```

PROGRAMME Ada

- Un programme Ada est constitué **d'unités de programme**, qui peuvent être
  - un paquetage (***package***)
  - un sous-programme (***function*** ou ***procedure***)
  - une unité générique, paquetage ou sous-programme
- Chaque unité est constituée
  - d'une partie spécification, visible aux autres unités
  - d'une partie élaboration du corps du programme, qui n'est pas visible
- Une unité importe une unité de bibliothèque, avec la clause de contexte:

  with *nom_simple_d_unité;*

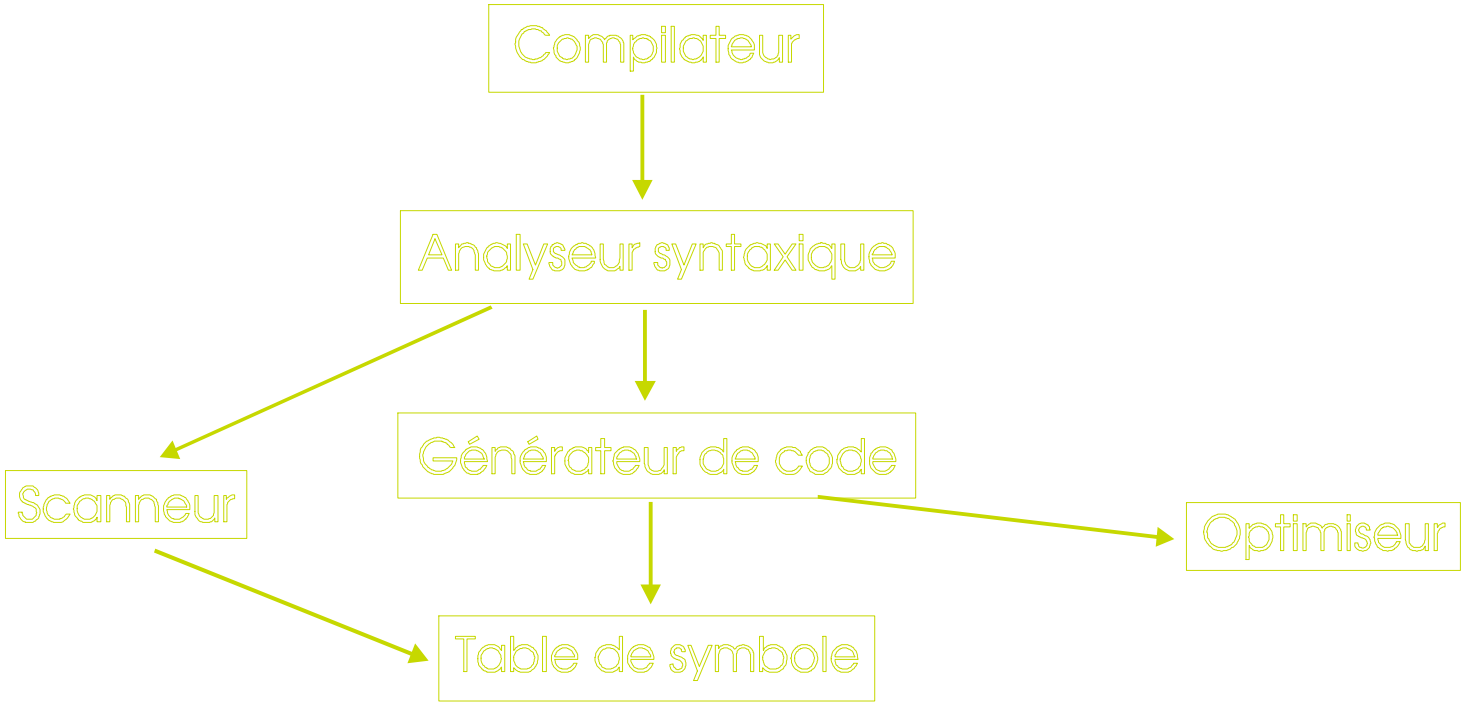
- La partie déclaration
  - spécification
  - élaborer les déclarations
- La partie instructions
  - corps
  - exécuter les instructions



PROGRAMME STRUCTURÉ À L'AIDE DE PAQUETAGE

- Les programmes Ada sont habituellement composés d'une collection de paquetages, qui contiennent une collection d'objets apparentés, e.g.
  - types,
  - variables,
  - objets,
  - sous-programmes,
  - classes.
- Les relations entre les paquetages peuvent être illustrées par un diagramme.
- La structure d'un programme Ada suit naturellement une structure de haut en bas (top-down design).

PROGRAMME STRUCTURÉ À L'AIDE DE PAQUETAGE



STRUCTURE DE PAQUETAGE (Unité de programme)

- La syntaxe d'une spécification de paquetage est:

spécification_de_paquetage ::=

package *identificateur* ***is***

{élément_déclaratif_de_base}

*[**private***

{élément_déclaratif_de_base}]

end *[identificateur];*

STRUCTURE DE PAQUETAGE (suite)

- La syntaxe du corps d'un paquetage est:

corps_de_paquetage ::=

package body *nom_simple_de_paquetage* ***is***
[partie déclarative]
*[**begin***
suite_d_instructions
*[**exception***
traite_d_exception]]
end *[nom_simple_de_paquetage];*

PAQUETAGES IMBRIQUÉS ET PARTAGÉS

- Les déclarations de paquetage peuvent être imbriqués à l'intérieur d'un autre paquetage, e.g. les paquetages génériques

  Ada.Text_IO.Integer_IO
  Ada.Text_IO.Modular_IO
  Ada.Text_IO.Float_IO
  Ada.Text_IO.Fixed_IO
  Ada.Text_IO.Decimal_IO
  Ada.Text_IO.Enumeration_IO

  sont imbriqués à l'intérieur du paquetage de la bibliothèque standard TEXT_IO (maintenant Ada.Text_IO)

- Avant d'utiliser un paquetage générique, il faut en créer une instance

  package *Int_IO* ***is new*** *Ada.Text_IO.Integer_IO (Integer);*

- Si un paquetage d'une bibliothèque est importé de plusieurs unités, ce paquetage est *partagé* plutôt que copié

Exemple

```
1   with Ada.Text_IO; USE Ada.Text_IO;
2
3   Procedure InstancierPaquetage is
4
5     Package Es_Reels   is new Ada.Text_IO.Float_IO(Float);
6     Package Es_Integer is new Ada.Text_IO.Integer_IO (Num => Integer);
7
8     choix     : String:="0";          --// un caractère
9     nbEntier  : Integer := 5 ;     --// un entier
10    nbReel    : Float := 10.00 ;   --// un reel
11
12  BEGIN
13    Put_line(Item => "-------------------------------------------------");
14    Put_line(Item => "*                    Test                       *");
15    Put_line(Item => "-------------------------------------------------");
16
17    New_line(2);
18
19    Put(Item => "La valeur de la variable nbEntier = ");
20    Es_Integer.Put(Item =>nbEntier);
21    New_line(1);
22    Put(Item => "La valeur de la variable nbReel = ");
23    Es_Reels.Put(Item =>nbReel);
24    New_line(5);
25    Put(Item => "Entrez votre choix ? :");
26    Get(Item => choix);  New_line(2);
27    Put_line(Item => "Le contenu de la variable choix est : "); Put_line(Item =>choix);
28
29  End InstancierPaquetage;
```

Résultat

```
 ----Hit any key to start.
------------------------------------------------
*                    Test                       *
------------------------------------------------


La valeur de la variable nbEntier =           5
La valeur de la variable nbReel =  1.00000E+01




Entrez votre choix ? :1


Le contenu de la variable choix est :
1

 ----Hit any key to continue.
```

PAQUETAGES PRÉINSTANCIÉS

- Pour les types prédéfinis *Integer, Float* et *Complex_Types* pour des programmes où la portabilité n'est pas importante, trois paquetages sont instanciés :
  - Ada.Integer_Text_IO
  - Ada.Float_Text_IO
  - Ada.Complex_Text_IO
  - Exemple :

```
with Ada.Integer_Text_IO;
use Ada.Integer_Text_IO;
```

- Unité de compilation d'Ada (déjà fournit)

```
with Ada.Text_IO;

pragma Elaborate_All (Ada.Text_IO);

package Ada.Float_Text_IO is
                    new Ada.Text_IO.Float_IO (Float);
```

Ada-Float_Text_IO.ads

ORDRE DE DÉCLARATION DES PAQUETAGES

- Le corps d'un paquetage doit être précédé de la spécification (de ce paquetage)
- Les unités de bibliothèque ne peuvent pas être importées avant d'avoir été spécifiées
- À l'intérieur d'un paquetage, les déclarations peuvent apparaître dans n'importe quel ordre en autant qu'elles ne sont pas utilisées avant d'être élaborées

ÉLABORATION DES PAQUETAGES

```
package P is
  function A return Integer;
end P;
```

P.ads

```
package body P is
  function A return Integer is
  begin
     return ….;
  end A;
end P;
```

P.adb

```
with P;
pragma Elaborate (P);
package Q is
  I : Integer := P.A;
end P;
```

Q.ads

SOUS-PROGRAMMES: Procédure et Fonction

- Les sous-programmes d'Ada sont similaires aux sous-programmes des autres langages de haut-niveau, sauf pour:
  - le mode de passage des paramètres
  - les associations de noms et de positions des arguments
  - les valeurs par défaut des paramètres actuels (arguments)
- Les sous-programmes Ada sont réentrants et peuvent être récursifs.

DÉCLARATION DE SOUS-PROGRAMME

```
function Max ( X, Y : in Integer) return Integer is
begin
    if ( X > Y) then
        return X;
    else
        return Y;
    end if;
end Max;
```

- Les sous-programmes déclarés dans le corps d'un paquetage ne requiert pas d'avoir un corps séparé.

SYNTAXE DU CORPS D'UN SOUS-PROGRAMME

- La syntaxe du corps d'un sous-programme est :

corps_de_sous-programme ::=
spécification_de_sous_programme ***is***
[partie déclarative]
begin
suite_d_instructions
*[****exception***
traite_d_exception]
end *[désignateur];*

- La liste des paramètres formels doit être identique à la liste de la déclaration correspondante.
- L'énoncé ***return*** *[expression*] peut être utilisé pour sortir du sous-programme.

SYNTAXE DU CORPS D'UN SOUS-PROGRAMME

- La syntaxe de la déclaration d'un sous-programme est :

déclaration_de_sous_programme ::= spécification_de_sous_programme
spécification_de_sous_programme ::=
procedure *identificateur [(paramètre {; paramètre})]*
| ***function*** *identificateur [(paramètre {; paramètre})]*
return *marque_de_type*

paramètre ::= liste_d_identificateur **:** ***in | out | in out***
marque_de_type [:= expression]

- Par exemple,

```
procedure G_Text ( Pt1, Pt2      : in Point;
                   Le_Texte      : in String);
```

OPÉRATEUR COMME NOM DE SOUS-PROGRAMME

- Ada permet d'utiliser un opérateur comme nom de sous-programme, e.g.

```
function "+" ( Left  : in Time;
               Right : in Duration) return Time;
```

PASSAGE DE PARAMÈTRES ET MODES

- La plupart des langages de haut-niveau utilise un ou les deux mécanismes suivants pour le passage de paramètres:
  - Appel par valeur ou appel par référence (adresse)

- Deux autres mécanismes existent :
  - Appel par résultat ou appel par valeur de résultat

- Le paramètre ***in*** d'Ada utilise le mécanisme *appel par valeur*
- Le paramètre ***out*** d'Ada utilise le mécanisme *appel par résultat*
- Le paramètre ***in out*** utilise soit l*e mécanisme appel par valeur de résultat,* ou le mécanisme *appel par référence*

PARAMÈTRES NOMMÉS ET POSITIONNELS

- Les déclarations de sous-programmes peuvent comporter des paramètres *formels* par défaut:

```
procedure ResetClock (
        hour      : in anHour;
        pm        : in Boolean := False;
        minute    : in aMinute := 0;
        second    : in aSecond :=0);
```

- À l'appel d'un sous-programme, on peut omettre les arguments (paramètres actuels) possédant des valeurs par défaut:

```
ResetClock ( my_hour);
    --- ResetClock (my_hour, 1) & ResetClock ( my_hour,,1) sont illégals
```

PARAMÈTRES NOMMÉS ET POSITIONNELS (suite)

- Les paramètres actuels peuvent être appariés par noms:

```
ResetClock ( hour     => my_hour);
ResetClock ( hour     => my_hour,
             minute   => 1);
```

- On ne peut utiliser de paramètres positionnels après avoir utiliser des paramètres par nom

TYPE DE DONNÉES ET DÉCLARATION D'OBJET

- La syntaxe de la déclaration d'un objet est :

déclaration_d_objet ::=
*liste_d_identificateur : [**constant**]*
identification_de_sous_type
[:= expression];

*liste_d_identificateur : [**constant**]*
définition_de_tableau_contraint
[:= expression];

INITIALISATION DE VARIABLE & CONSTANTS

- Les valeurs initiales peuvent être des expressions
- Les constantes sont initialisées comme des variables:

```
Pi                  : constant          := 3.141_592_653;
Limite              : constant Integer   := 100;
Limite_Inférieure   : constant Integer   := Limite / 100;
```

- Un bon style de programmation encourage la déclaration des constantes, et l'élimination des "nombres magiques" dans les expressions.

ATTRIBUTS

- Chaque entité d'Ada, incluant les objets, les types et les tâches, possède des attributs, auxquels on accède de la façon suivante:

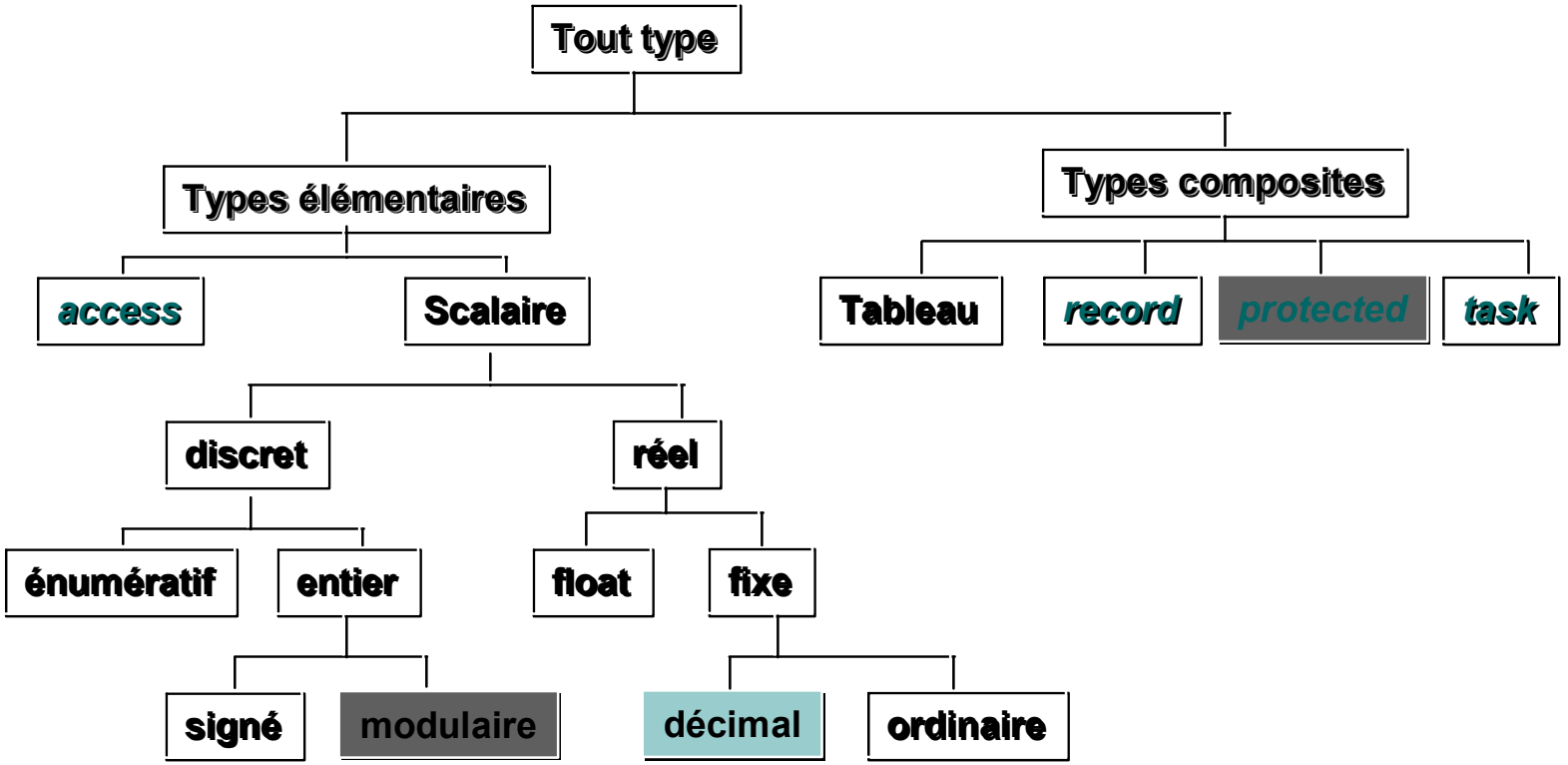
  nom_d_entité'Nom_D_Attribut

- Les attributs prédéfinis d'Ada sont à l'annexe A du manuel de référence et incluent :
  - *Address* pour n'importe quel objet
  - *First* pour les types scalaires
  - *Range* pour les types discrets

- L'utilisation d'attributs améliore la lisibilité et la portabilité d'un programme.

TYPE DE DONNÉES

- Les types de données sont définis hiérarchiquement de la façon suivante:



- Tous les types doivent être nommés.

DÉCLARATION DE TYPE

- Une déclaration de type est un squelette (moule) pour les données
- Ada fournit plusieurs types scalaires prédéfinis: *Integer*, *Float, Boolean*, et *Character*
- Ada fournit deux catégories de types composés:
  - tableaux (***array***)
  - enregistrement (***record***)
- La syntaxe d'une déclaration de type est:

déclaration_type::=

type *identificateur_type* ***is*** *constructeur_type;*

TYPE DE DONNÉE SCALAIRE & ÉNUMÉRATIF

- Les types:

  Integer

  Float

  sont prédéfinis dans le paquetage Standard, avec les sous-types

  Natural

  Positive

- Les entiers (*Integer*) et les réels (*Float*) sont définis comme des sous-types des
  - entiers universels (universal_integer)
  - réels universels (universal_real)

  dans le paquetage Standard

TYPES UNIVERSELS

- Les types universels ont une plage (***range***) et une précision sans limite

- Les variables ne peuvent être déclarées de type universel, seulement les littéraux numériques sont de type universel

TYPE ÉNUMÉRATIF

- Une définition de type énumératif définit un type énumératif,

  type *identificateur_de_type* ***is*** *(littéral_d_énumération*
  {, littéral_d_énumération});

- Un littéral d'énumération peut être un identificateur ou un littéral caractère :

  type *Jour* ***is*** *(Lun, Mar, Mer, Jeu, Ven, Sam, Dim);*
  type *Délimiteur* ***is*** *('&', ''', '(', ')', ...);*
  type *T_Jour* ***is*** *(Lun, Mar, Mer, Jeu, Ven, Sam, Dim);*
  type *T_Délimiteur* ***is*** *('&', ''', '(', ')', ...*

- Opérations sur les types énumératifs:
  - attributs *Succ* et *Pred*
  - attributs *First* et *Last*
  - attributs *Pos* et *Val*
  - attributs *Image* et *Value*

TYPES ÉNUMÉRATIFS PRÉDÉFINIS

- Le paquetage STANDARD définit les types énumératifs *Boolean* et *Character*.

```
type Boolean     is (False, True);
type Character is ( nul, soh,  ....'.', '!', '0', '1', .....
                                    'A', 'B', ........
                                    '~', del);
```

- Les types énumératifs doivent être utilisés au lieu de listes de constantes:

```
LUN : constant  INTEGER := 1;
MAR: constant  INTEGER := 2;
"
```

TYPE ÉNUMÉRATIF (Attributs)

```
with Ada.TEXT_IO;
use Ada.TEXT_IO;

procedure TEST2 is
   type Jour is (Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche);
   Fin_Semaine : constant String := Jour'Image(Samedi) & " " &
                                    Jour'Image(Dimanche);
begin
   Put ("Jour'First =  "); Put (Jour'Image(Jour'First)); New_Line;
   Put ("Jour'Last  =  "); Put (Jour'Image(Jour'Last)); New_Line;
   Put ("Jour'Succ(Lundi)  =  "); Put (Jour'Image(Jour'Succ(Lundi))); New_Line;
   Put ("Jour'Pred(Dimanche)  =  "); Put (Jour'Image(Jour'Pred(Dimanche))); New_Line;
   Put ("Jour'Pos(Lundi)  =  "); Put (Integer'Image(Jour'Pos(Lundi))); New_Line;
   Put ("Jour'Val(4)  =  "); Put (Jour'Image(Jour'Val(4))); New_Line;
   Put ("Jour'Value(""Mercredi"")  = "); Put (Jour'Image(Jour'Value("Mercredi"))); New_Line;
   Put ("Jour'Image(Mercredi)  = "); Put (Jour'Image(Mercredi)); New_Line;
   Put ("Fin de semaine= "); Put (Fin_Semaine ); New_Line;
   Put ("Samedi = "); Put (Fin_Semaine (1..6)); New_Line;
end TEST2;
```

TYPE ÉNUMÉRATIF (Attributs)

```
 with Ada.TEXT_IO;
use Ada.TEXT_IO;
procedure TEST21 is
     type Jour is (Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche);
      Fin_Semaine : constant String := Jour'Image(Samedi) & " " & Jour'Image(Dimanche);
begin
      Put_Line ("Jour'First =  "                    & Jour'Image(Jour'First) );
      Put_Line ("Jour'Last  =  "                    & Jour'Image(Jour'Last) );
      Put_Line ("Jour'Succ(Lundi)  =  "   & Jour'Image(Jour'Succ(Lundi)));
      Put_Line ("Jour'Pred(Dimanche)  =  "          & Jour'Image(Jour'Pred(Dimanche)));
      Put_Line ("Jour'Pos(Lundi)  =  "    & Integer'Image(Jour'Pos(Lundi)));
     Put_Line ("Jour'Val(4)  =  "                   & Jour'Image(Jour'Val(4)));
     Put_Line ("Jour'Value(""Mercredi"")  = "       & Jour'Image(Jour'Value("Mercredi")));
      Put_Line ("Jour'Image(Mercredi)  = "          & Jour'Image(Mercredi));
      Put_Line ("Fin de semaine= "                  & Fin_Semaine );
      Put_Line ("Samedi = "                         & Fin_Semaine (1..6));
end TEST21;
```

TYPE ÉNUMÉRATIF (Attributs)

- Résultats

```
Jour'First =  Lundi
Jour'Last  =  Dimanche
Jour'Succ(Lundi)  =  Mardi
Jour'Pred(Dimanche)  =  Samedi
Jour'Pos(Lundi)  =  0
Jour'Val(4)  =  Vendredi
Jour'Value("Mercredi")  = Mercredi
Jour'Image(Mercredi)  = Mercredi
Fin de semaine= Samedi Dimanche
Samedi = Samedi
```

TYPE ÉNUMÉRATIF (Attributs)

```
with Ada.TEXT_IO, SYSTEM;
use Ada.TEXT_IO;

procedure TEST3 is
   Un_Caractère: Character;
begin
   Un_Caractère:= Character'First;
   while ( Un_Caractère/= Character'Last) loop
      Put (Character'Image(Un_Caractère));Put("    "); Put (Un_Caractère); New_Line;
      Un_Caractère:= Character'Succ(Un_Caractère);
   end loop;
end TEST3;
```

TYPE ÉNUMÉRATIF (Character): Ada.Characters.Latin_1 (ISO-8859-1)

```
with TEXT_IO, SYSTEM;
use TEXT_IO;
with Ada.Characters.Latin_1;
use Ada.Characters.Latin_1;
procedure TEST3 is
   unCaractère: Character := Character'First;
begin
   while (unCaractère/= Character'Last) loop
      Put (Character'Image(unCaractère));Put("    "); Put (unCaractère); New_Line;
      unCaractère:= Character'Succ(unCaractère);
   end loop;
end TEST3;
```

TYPE ÉNUMÉRATIF (Character): Ada.Characters.Latin_1 (ISO-8859-1)

```
with TEXT_IO;
use TEXT_IO;
with Ada.Characters.Latin_1;
use Ada.Characters.Latin_1;
procedure TEST4 is
begin
   for unCaractère in Character'Range loop
      Put_Line (Character'Image(unCaractère) & "    "  & unCaractère);
   end loop;
end TEST4;
```

TYPES COMPOSÉS-- ENREGISTREMENT (ARTICLE)

- Un enregistrement (***record***) est un type composé analogue à celui de PASCAL, mais plus flexible.

- La syntaxe est:

définition_de_type_article ::=
record
déclaration_de_composant
{déclaration_de_composant}
end record;

déclaration_de_composant ::=
liste_d_identificateur : indication_de_sous_type
[:= expression];

DÉCLARATION: TYPE ARTICLE

```
type Point is
    record
      X : Float := 0.0;
      Y : Float := 0.0;
    end record;
type Fenêtre is
     record
       Coin_Gauche : Point;
       Coin_Droit     : Point := (640.0, 480.0);
     end record;

     AXE  : Fenêtre;
     P, Q : Point;

 begin
   P.X := .....
   P.Y  := .....
```

DÉCLARATION: TYPE ARTICLE

- Les champs dans un enregistrement peuvent être initialisés

```
type Point3D is
   record
      X : Float := 0.0;
      Y : Float := 0.0;
      Z : Float := 0.0;
   end record;
```

TABLEAU (ARRAY)

- La syntaxe de la déclaration d'un type tableau est:

déclaration_de_type_tableau ::=
type *identificateur_de_type* ***is***
array *(intervalle_discret { , intervalle_discret})*
of *type_de_composant;*

- Un intervalle_discret peut être:
  - expression .. expression
  - nom_de_type_discret (e.g. Un_Jour)

EXEMPLES DE TABLEAUX

- Tableaux unidimensionnels

```
type Vecteur     is array ( 1 .. 10 ) of Integer;
Première_Col : constant Integer := 1;
Dernière_Col :  constant Integer := 132;
type Ligne       is array ( Première_Col .. Dernière_Col)
                                              of Character;
```

- Tableaux multidimensionnels

```
Première_Ligne   : constant Integer := 1;
Dernière_Ligne   : constant Integer := 66;
type Une_Page is array
    (Première_Ligne .. Dernière_Ligne)    of Ligne;
     --    ou
type Une_Autre_Page is array
                    (Première_Ligne .. Dernière_Ligne,
          Première_Col .. Dernière_Col) of Character;
```

EXEMPLES DE TABLEAUX

- On référence un élément d'un tableau par les indices de cet élément.

```
La_Ligne            : Ligne;
La_Page             : Une_Page;
I                   : Integer                 :=
  Première_Ligne;
J           : Integer                 := Dernière_Ligne;
L_Autre_Page        : Une_Autre_Page;
begin
La_Ligne(5)                   := 'c';
La_Page (I)                   := La_Ligne;
La_Page(I)(Première_Col)      := '1';
L_Autre_Page ( i, j)          := '7';
```

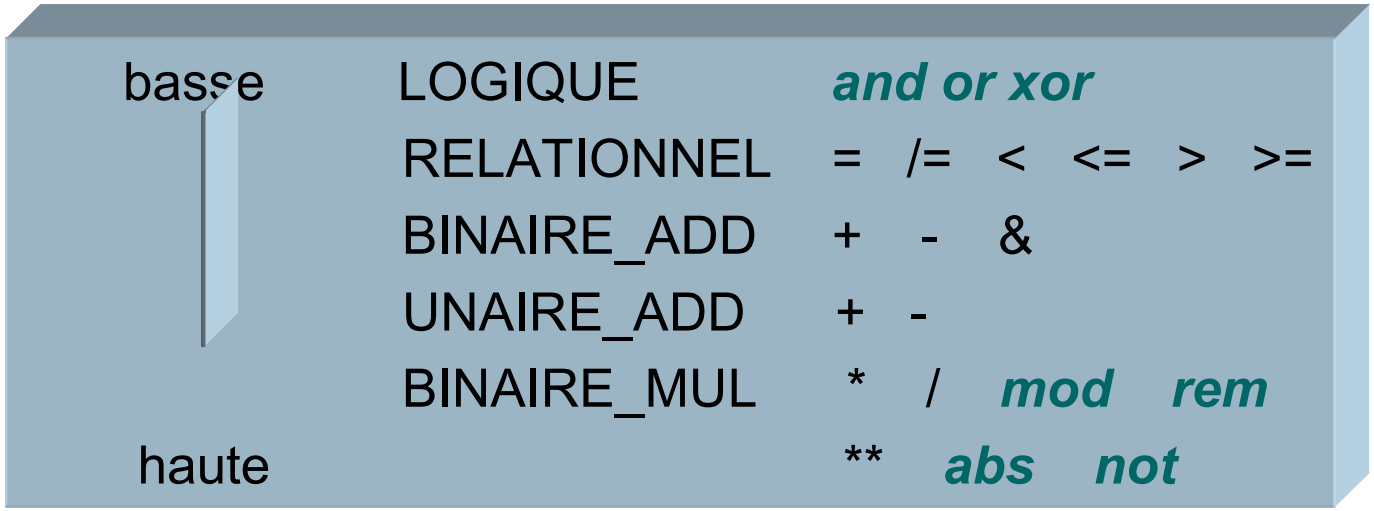
EXPRESSIONS

- Les expressions sont comme dans les autres langages évolués sauf :
  - Ada possède des opérateurs pour les tableaux (booléens),
  - Ada utilise le contexte pour résoudre les expressions surchargées (overloaded)
- Ada ne permet pas les expressions en mode mixte

```
4.0;                                  -- expression réelle
2.0 * Pi;
B * B - Float (4) * A * C;
Voiture (1 .. 3 ) = "BMW" ;    -- expression relationnelle
```

CLASSES D'OPÉRATEURS

- Les opérateurs sont divisés selon les classes suivantes:



- La règle de précédence de gauche à droite est utilisée lorsque les opérateurs ont la même priorité.

X	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8
X **mod** 4	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0
X **mod** (-4)	0	-3	-2	-1	0	-3	-2	-1	0	-3	-2	-1	0	-3	-2	-1	0
X **rem** 4 X **rem** (-4)	0	-3	-2	-1	0	-3	-2	-1	0	1	2	3	0	1	2	3	0

OPÉRATEURS LOGIQUES

- Les opérateurs logiques peuvent être appliqués à des tableaux *Boolean* de même longueur.

```
Filtre  :  array (1 .. 30 ) of Boolean;
Filtre(1 .. 10) and Filtre ( 15 .. 24 )
```

- Ada possède aussi la forme de contrôle en raccourci

  and then

  or else

```
if n = 0 or else A(n) = Valeur_Trouvée then
    ......
end if;
```

OPÉRATEURS RELATIONNELS & APPARTENANCE

- Les opérateurs d'ordre sont prédéfinis pour tout type scalaire et pour tout type tableau discret.

```
X /= Y
"" < "A" and "A" < "AA"    -- true
```

- Ada possède aussi des opérateurs d'appartenance
  - **in**
  - **not in**
- **Exemple**

```
n not in 1 .. 10
Aujourdhui in Lundi .. Vendredi
I in X'Range
if ( Index_Block not in DiskMap'Range ) or else
      ( not FREE_MAP(INDEX_BLOCK)) then   .....
end if;
```

OPÉRATEURS RELATIONNELS & APPARTENANCE

```
with Ada.Text_IO;
procedure Test123 is
   package Flt_IO is new Ada.Text_IO.Float_IO ( Float);
   use Flt_IO;
   use Ada.Text_IO;
   Limite_Inférieure : constant :=  1.0;
   Limite_Supérieure : constant := 56.0;
   X : Float := 11.0;
begin

    if X in Limite_Inférieure .. Limite_Supérieure then
       Put ("X = ");
       Put (Item => X, Fore => 5, Aft => 2, Exp => 0);
       Put (" est dans l'intervalle ");
       Put ( Limite_Inférieure, Fore => 5, Aft => 2, Exp => 0); Put (" et ");
       Put ( Limite_Supérieure, Fore => 5, Aft => 2, Exp => 0); New_Line;
    end if;
end Test123;
```

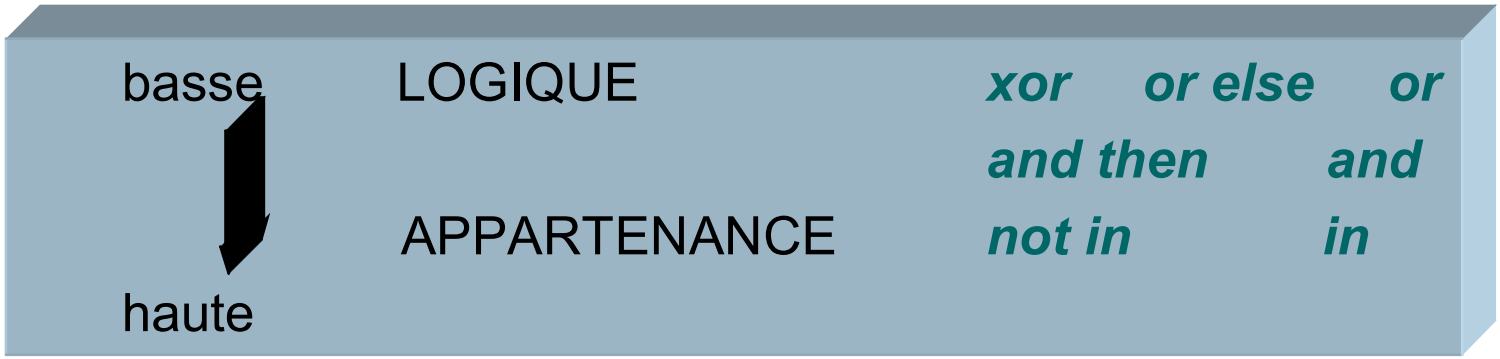
X = 11.00 est dans l'intervalle 1.00 et 56.00

```
procedure Put (    Item : in Num;
                   Fore : in Field := Default_Fore;
                   Aft  : in Field := Default_Aft;
                   Exp  : in Field := Default_Exp);
```

Fore.Aft E Exp

AUTRES D'OPÉRATEURS

- Les opérateurs supplémentaires et de raccourci sont :



ÉNONCÉS

- Un ';' à la fin de chaque énoncé pour éliminer l'ambiguïté.
- Énoncé ***exit*** dans les boucles (***loop***).
- Un énoncé ***null***.

AFFECTATION

- En plus des affectations simples de même type, Ada permet
  - les affectations d'articles (enregistrement)

```
Point_1 := Point_2;
```

  - les affectations de tableaux et de tranches

```
Ligne_1          := Ligne_2;
Ligne(1 .. 10 )  := "0123456789";
Ligne(1 .. 4)    := Ligne(3 .. 6);
```

ÉNONCÉS CONDITIONNELS & BOUCLES

- Ada possède 2 énoncés conditionnels : ***if*** et ***case;***
- Ada possède une seule boucle avec les clauses ***while*** et ***for***.

ÉNONCÉS if

- La syntaxe de l'énoncé ***if*** est:

```
instruction_if  ::=
        if condition then
            suite_d_instructions
        {elsif condition then
            suite_d_instructions}
         {else
            suite_d_instructions}
          end if;
```

ÉNONCÉS if (suite)

- Exemple:

```
if Mois = Décembre and Jour = 31 then
   Mois        := Janvier;
   Jour        := 1;
    Année      := Année + 1;
end if;

if Ligne_Trop_Courte then
    raise Fin_De_Ligne;
elsif Ligne_Pleine then
    New_Line;
    Put(Élément);
else
    null;
end if;
```

ÉNONCÉS if (suite)

- Exemple:

```
type T_Date is
record
    Jour        : T_Jour;
    Mois        : T_Mois;
    Année       : T_Année;
end record;
Date : T_Date := ( 10, Décembre, 2005);
if Date.Mois = Décembre and then Date.Jour = 31 then
   Date := ( 01, Janvier, Année + 1);
-- Date := ( T_Jour'First, T_Mois'First, Année + 1);
end if;
```

ÉNONCÉS case

- La syntaxe de l'énoncé ***case*** est:

```
instruction_case ::=
    case expression is
        when choix { | choix }  =>
            liste_d_énoncés
        {when choix { | choix } =>
            liste_d_énoncés
     end case;
```

- Le choix peut être:
  - littéral
  - intervalle discret
  - ***others***

ÉNONCÉS case (suite)

- Exemple:

```
case Capteur   is
   when Élévation        => Noter_Élévation (Valeur_Capteur);
   when Azimut                    => Noter_Azimut    (Valeur_Capteur);
   when Distance         => Noter_Distance   (Valeur_Capteur);
   when others           => null;
end case;

case Aujourdhui is
   when Lundi            => Heure_D_Ouverture := (12, 18);
   when Mardi..Mercredi           => Heure_D_Ouverture := (9, 18);
   when Jeudi | Vendredi => Heure_D_Ouverture := (9, 21);
   when Samedi           => Heure_D_Ouverture := (9, 17);
   when others           => Heure_D_Ouverture := (0, 0);
end case;
```

ÉNONCÉS case (suite)

- Exemple:

```
type T_Année    is range 1900 .. 2100;
type T_Jour     is range 1 .. 31;
type T_Mois     is
    (Janvier,   Février,   Mars,      Avril,
     Mai,       Juin,     Juillet,   Août,
     Septembre, Octobre,  Novembre,  Décembre);
ceMois          : T_Mois;
cetteAnnée      : T_Année;
nombreDeJours   : T_Jour;

case ceMois is
   when Février =>
     if (cetteAnnée mod 4 = 0) and
         ((cetteAnnée mod 100 /= 0) or (cetteAnnée mod 400 = 0 )) then
       nombreDeJours := 29;
     else
       nombreDeJours := 28;
     end if;
   when Avril | Juin | Septembre | Novembre =>
     nombreDeJours := 30;
   when Janvier | Mars | Mai | Juillet | Août | Octobre | Décembre =>
     nombreDeJours := 31;
end case;
```

ÉNONCÉ loop

- La syntaxe d'un énoncé ***loop*** est:

instruction_loop ::=
[nom_simple_de_boucle:]
[schéma_d_itération] ***loop***
suite_d_instructions
end loop*;*

schéma_d_itération ::= ***while*** *condition*
| ***for*** *identificateur_de_boucle* ***in*** *[****reverse****]*
intervalle_discret

EXEMPLE DE BOUCLE

- La clause ***for***

```
for I in X'Range loop
   X(I) := A * X(I) + B;
end loop;
```

- La clause ***while***

```
while not Délimiteur loop
  i := i + 1;
end loop;
```

```
for I in X'First..X'Last loop
   X(I) := A * X(I) + B;
end loop;
```

```
for i in 1 .. n loop
   X(i) := A * X(i) + B;
end loop;
```

SORTIE DE BOUCLE

- L'énoncé ***exit*** est utilisé pour sortir d'une boucle (après ***end loop*** ou à la boucle étiquetée)

  *instruction_exit ::= **exit** [nom_de_boucle] [**when** condition];*

- Exemple:

```
SAUTER:   loop
                Get (c);
                exit when      ( c /= ' ' and then
                                 c /= Ada.Characters.Latin_1.Tab);
          end loop;
```

Eh oui! goto

- La syntaxe de l'instruction ***goto*** est:

  *instruction_goto ::= **goto** étiquette;*

- Une étiquette <<étiquette>>

  goto Premier_De_Classe;

  << Premier_De_Classe >>

  .....