Typage

- Le typage des données permet de mieux structurer le code et de le vérifier (partiellement) sans l'exécuter

- Un **type de donnée** est défini par un ensemble de valeurs et un ensemble d'opérations pouvant être effectuées sur ces données
  - *type booléen : 2 valeurs, opérations logiques*
  - *type entier : un intervalle de N, opérations arithmétiques*
  - *type Personne décrit par un nom et un prénom : tous les couples de chaines de caractères, opérations de création, modification des noms et prénoms*
  - *...*

- Certains langages ne sont pas typés : *Lisp, Prolog, Smalltalk, ...*

- La plupart le sont : *C, Pascal, Java, ...*

- Les premiers langages ne permettaient pas de créer de nouveaux types (*Algol*, *Cobol*, *Fortran*)

Contrôle de type

- Le contrôle de type vérifie qu'une opération apparaissant dans une instruction porte bien sur des valeurs adéquates

- Un langage est dit fortement typé si :
  - chaque objet appartient à un seul type
  - le type d'une expression peut être déterminé syntaxiquement à la compilation
  - il n'y a pas de conversion implicite

- ADA est (quasiment) le seul langage fortement typé

- Un langage fortement typé permet :
  - de vérifier le bon usage des variables et opérations dans un programme sans l'exécuter
  - d'écrire des programmes plus robustes
  - de négliger certains tests de type dans le code (programmes plus efficaces)

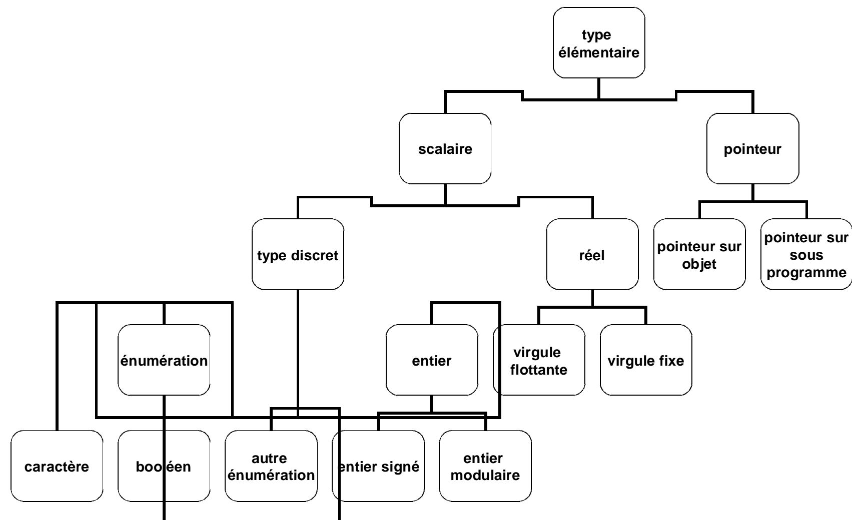
Types en ADA (1/2)

- Règles sur les types :
  - les opérandes d'une même opération doivent être du même type
  - dans une affectation, le type de la variable et celui de l'expression doivent être les mêmes
  - les types d'un paramètre formel et celui du paramètre effectif doivent être les mêmes
- Pas de conversion implicite :
  - affecter un entier à une variable de type flottant nécessite une conversion explicite!
  - permet d'éviter les bugs liés à des conversions non maîtrisées
  - *cas particulier* : aucune conversion n'est nécessaire entre un type et un de ses sous-types
- Contrôle de type dynamique :
  - ADA contrôle dynamiquement les types à l'exécution en s'assurant qu'une valeur donnée à une variable est bien dans le domaine du type de la variable
  - Ces tests peuvent être supprimés dans le logiciel final pour optimisation par la directive `pragma Suppress (All_Checks)`

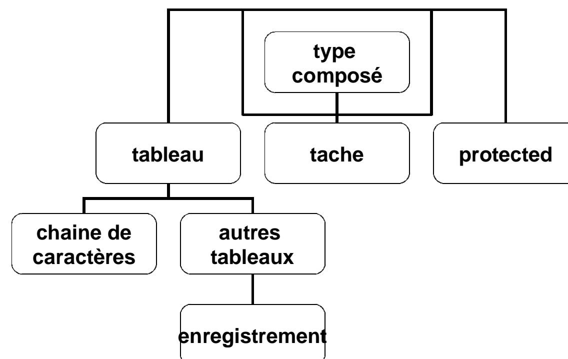
Types en ADA (2/2)

- ADA définit peu de types de base :
  - `Character`
  - `String`
  - `Boolean`
  - `Integer`
  - `Float`
  - `Access` (pointeur)
- ADA permet de définir :
  - des sous-types (ou restriction de type) : sous ensemble des valeurs d'un type
  - des types dérivés : analogues aux sous-types mais incompatibles avec leur type parent
  - des types numériques modulaires
  - des types énumératifs
  - des types composites : tableaux, record
  - des types synonymes
  - des types à discriminant : la détermination du type dépend d'un paramètre

Types élémentaires en ADA



Types composés en ADA



Les tâches sont des types car elles sont déclarées comme n'importe quel objet. Le type `protected` sert à déclarer des ressources (données ou sous-programmes) partagées entre plusieurs tâches.

Attributs en ADA

- En ADA, un attribut est un mécanisme permettant de récupérer certaines caractéristiques d'une entité (variable, type, ...). Un attribut est accédé par le nom de l'entité suivi d'une apostrophe et du nom de l'attribut.

- *Exemple : l'attribut* BASE *appliqué à un type ou un sous-type donne le type de base du type ou du sous-type. L'attribut* LAST *appliqué au type* INTEGER *donne le plus grand entier représentable. Le code suivant écrit 2147483647 (2^{31} - 1) à l'écran.*

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

procedure Test is

subtype int is INTEGER range 0..100;

begin
   put(int'BASE'LAST);
end;
```


Opérations sur les types

- Pour les types prédéfinis, ou les autres, les opérations toujours définies sont (liste non exhaustive) :
  - l'affectation
  - l'égalité et l'inégalité
  - l'attribut STORAGE_SIZE qui, appliqué au type donne la place occupée en mémoire par le plus grand objet de ce type en nombre d'unités mémoire (octets).
  - l'attribut SIZE est identique à STORAGE_SIZE mais donne l'occupation mémoire en bits. Appliqué à un objet, il donne l'occupation mémoire de l'objet.
  - l'attribut ADRESS appliqué à un objet donne son adresse en mémoire.
  - l'attribut BASE appliqué à un type ou un sous-type donne le type de base du type ou du sous-type.

- Il est possible d'interdire l'usage des opérations d'affectation et d'égalité/inégalité dans les *types limités*

Déclaration de type complet

- Les déclarations de type définissent généralement des types nommés. Dans certains cas (déclaration interne à une autre déclaration) on peut avoir des types anonymes.

full_type_declaration ::= **type** defining_identifier [known_discriminant_part] **is** type_definition ;
| task_type_declaration
| protected_type_declaration

type_definition ::= enumeration_type_definition | integer_type_definition | real_type_definition
| array_type_definition | record_type_definition | access_type_definition
| derived_type_definition | interface_type_definition

- *Exemples :*

```
type Color is (Vert, Bleu, Rouge, Jaune, Violet, Blanc, Noir);

type Table is array(1 .. 12) of Integer;
```


Déclaration de sous-type

- Déclarer un sous-type ne crée pas un nouveau type : il s'agit d'une restriction d'un type existant, restriction exprimée par une contrainte.

subtype_declaration ::= **subtype** defining_identifier **is** subtype_indication ;

subtype_indication ::= [null_exclusion] subtype_mark [constraint]

subtype_mark ::= *subtype*_name

constraint ::= scalar_constraint | composite_constraint

scalar_constraint ::= range_constraint | digits_constraint | delta_constraint

composite_constraint ::= index_constraint | discriminant_constraint

- *Exemples :*

```
subtype Natural is Integer range 0.. Integer'LAST;
subtype Entier is Integer;
subtype Square    is Matrix(1 .. 10, 1 .. 10);
subtype Male      is Person(Sex => M);
subtype Binop_Ref is not null Binop_Ptr;
```

Contraintes de sous-type

- Les contraintes de sous-types sont contrôlées à la compilation lorsque la valeur affectée à une variable de ce sous-type est explicite (warning).
- *Exemple : la compilation de ce programme produit un warning et son exécution est stoppé par la levée d'une exception CONSTRAINT_ERROR*

```
procedure Test is

subtype int is INTEGER range 0..100;
i : int;

begin
   i := 110;
end;
```

- Lorsque le contrôle n'est pas possible à la compilation (valeur non explicite), une exception sera quand même levée à l'exécution en cas de violation des contraintes.
- *Exemple : ce programme compile sans problème mais son exécution est stoppé par la levée d'une exception CONSTRAINT_ERROR*

```
procedure Test is

subtype int is INTEGER range 0..100;
i : int; var : integer := 110;

begin
   i := var;
end;
```


Déclaration d'objet

- ADA permet la déclaration et l'instanciation dans la même instruction. Il est possible également d'introduire des sous-types anonymes.

```
object_declaration ::= defining_identifier_list : [ aliased ] [ constant ] subtype_indication [ := expression] ;
                     | defining_identifier_list : [ aliased ] [ constant ] access_definition [ := expression ] ;
                     | defining_identifier_list : [ aliased ] [ constant ] array_type_definition [ := expression ] ;
                     | single_task_declaration
                     | single_protected_declaration

defining_identifier_list ::= defining_identifier { , defining_identifier }
```

- `aliased` indique que l'objet déclaré peut être référencé (on récupère un pointeur dessus par l'attribut `ACCESS`).
- *Exemples :*

```
var : Integer := 12;
size : aliased Integer range 0..1000 := 0;
pi : constant Float := 3.1415927;
a,b : Float := 2/3;
```

Types dérivés (1/3)

- Un type dérivé est un **nouveau type** créé à partir d'un type existant (type parent). Un type dérivé est *incompatible* avec son type parent, contrairement à un sous-type.

derived_type_definition ::= [**abstract**] [**limited**] **new** *parent*_subtype_indication ;

- Un type dérivé peut servir à éviter des confusions entre entités prenant leurs valeurs dans le même ensemble mais de natures différentes.

- *Exemple : on veut gérer dans un programme des sommes en euros et en francs, sommes représentées dans les deux cas par des réels*

```
type Euro is new Float;
type Franc is new Float;
sEuro : Euro := 20.0; sFranc : Franc := 50.0; s : Float := 10.0;

sEuro := sEuro + s; -- interdit par le compilateur
sEuro := sEuro + sFranc; -- interdit par le compilateur
```


Types dérivés (2/3)

- Il est possible de dériver un type en posant une contrainte, c'est-à-dire dériver un sous-type anonyme.

- *Exemple :*

```
type Naturel is new Integer range 0..Integer'LAST;
```

- Un type dérivé hérite des opérations et fonctions définies dans le paquetage où le type parent est défini.

- Si un type dérivé est défini comme abstrait (`abstract`), il ne peut exister d'objet de ce type, qui ne sert que comme type parent d'autres types.

Types dérivés (3/3)

- Si un type dérivé est défini comme limité (**`limited`**) l'utilisation de l'affectation et des opérateurs d'égalité et d'inégalité sur des objets de ce type est interdite.

- *Intérêt du type limité* : empêcher des affectations globales de types composés qui ne copient pas les valeurs des champs ou redéfinir l'égalité/inégalité entre objets composés.

- *Exemple :*
  - *on veut que l'affectation d'une variable de type liste effectue une copie des éléments de la liste*
  - *on veut que deux objets de type Personne soient égaux s'ils ont le même numéro de sécu*

```
type Personne is limited record
   ID : Integer;
   Nom : String;
   Date_Naissance : Date;
   Profession : String;
end record;
```


Renommage

- On peut renommer un type en le dérivant : `type Truc is new Machin;`

- ADA permet en fait de renommer différents objets désignés par un identificateur, ce qui permet d'améliorer la lisibilité des programmes. Le renommage ajoute un surnom, mais ne supprime pas le nom précédent.

- Renommage des variables :

object_renaming_declaration ::= defining_identifier **:** [null_exclusion] subtype_mark **renames** *object*_name;
| defining_identifier **:** access_definition **renames** *object*_name;

- *Exemple :* `rm : Personne renames Roger_Martin;`

Types scalaires

- Les nombres, caractères, booléens et énumérations sont des types scalaires (types à valeurs simples et ne donnant accès à aucun autre objet)
- Un ordre total existe sur chacun de ces types : les valeurs sont ordonnées et peuvent être comparées par les opérateurs relationnels
- les opérations sur *les valeurs* d'un type scalaire sont l'affectation, =, /=, <=, <, >= et >
- Attributs communs à tous les types scalaires :
  - **`FIRST`** renvoie la plus petite valeur du type
  - **`LAST`** renvoie la plus grande valeur du type
- *Exemples :* **`Character'FIRST`** *vaut* **`'a'`**, **`Boolean'LAST`** *vaut* **`True`**.

Types discrets

- Les types discrets sont les types scalaires énumératifs (booléen et caractère) et les entiers.
- Les types discrets possèdent des attributs "fonctions" à un paramètre :
  - **`POS`** donne le code interne de l'objet passé en paramètre
  - **`VAL`** donne l'objet dont le code est passé en paramètre
  - **`SUCC`** donne l'objet suivant le paramètre
  - **`PRED`** donne l'objet précédent le paramètre
  - **`IMAGE`** donne la représentation en chaine de caractère de la valeur passée en paramètre (écriture habituelle pour les entiers, caractères entre apostrophes, ...)
  - **`WIDTH`** donne la longueur de la plus longue chaine de caractères représentant un objet du type
  - **`VALUE`** est le contraire de IMAGE
- *Exemples :* **`Character'POS('A')`** *vaut* **`65`**, **`Character'VAL(65)`** *vaut* **`'A'`**, **`Boolean'SUCC(False)`** *vaut* **`True`**, **`Boolean'SUCC(True)`** lève une **`CONSTRAINT_ERROR`**, **`Integer'IMAGE(23)`** vaut **`" 23"`**

Type énumérés

- Un type énuméré est un type dont on décrit in extenso l'ensemble des valeurs. Les valeurs d'une énumération sont énumérées à partir de 0.

```
enumeration_type_definition ::= ( enumeration_literal_specification { , enumeration_literal_specification }
   )

enumeration_literal_specification ::=  defining_identifier | defining_character_literal

defining_character_literal ::= character_literal
```

- *Exemples* :

```
type Jour is (Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche);
type Voyelle is ('a','e','i','o','u','y');

subtype Jour_de_semaine is Jour range Lundi.. Vendredi;

-- Jour'POS(Lundi) vaut 1
-- Jour'SUCC(Mercredi) vaut Jeudi
-- Jour'IMAGE(Jour'VAL(2)) vaut Mercredi
```


Boolean

- **Boolean** est un type énuméré de valeurs **False** et **True**

```
type Boolean is (False,True);
```

- En plus des opérations sur les types énumérés, **Boolean** possède les opérateurs logiques **and, or, xor et not** (**not** est prioritaire sur les 3 autres).

- *Expressions booléennes* : ADA impose l'usage de parenthèses quand plusieurs opérateurs logiques apparaissent dans une expression.

- *Exemples* :

```
A or Y and Z -- refusé par le compilateur
```

Character

- **Character** est un type énuméré dont les valeurs correspondant aux valeurs numérotées de 0 à 255 du code LATIN-1 de la norme ISO/IEC 10646:2003 (dont les 128 premières valeurs sont celles du code ASCII).

- Les caractères sont écrits entre apostrophes

- On peut redéfinir son propre jeu de caractère :
  - en créant un sous-type d'intervalle :

```
subtype Car is Character range 'a'..'Z';
```

  - en définissant un nouveau type énuméré :

```
type Hexa is ('0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F');
```


Integer (1/2)

- **Integer** est un type discret dont les valeurs vont de **Integer'FIRST** à **Integer'LAST**

- En plus des opérateurs sur les types discrets, **Integer** possède des opérateurs arithmétiques :
  - opérateurs unaires : **+**, **-**, **abs**
  - opérateurs binaires : **+**, **-**, *****, **/**, ******, **rem**, **mod**

- Il existe d'autres types entiers : **Short_Integer** (intervalle plus réduit que **Integer**) et **Long_Integer** (intervalle plus grand que **Integer**)

- Les limites de représentation des entiers sont données par **System.MIN_INT** et **System.MAX_INT**

Integer (2/2)

- Définition d'un nouveau type entier signé :

```
type Page_Num  is range 1 .. 200;
```

- Cette définition équivaut à définir un type dérivé et un sous type :

```
type Temp is new Integer;
subtype Page_Num is Temp range 1..200;
```

- Il est possible de définir un type entier modulaire :

```
type Heure is mod 24;
```

- Définition d'un sous-type de **`Integer`** :

```
type Coordonnees is Integer range -50..50;
```


Float (1/2)

- **`Float`** est un type scalaire représentant les réels dont les valeurs vont de **`Float'FIRST`** à **`Float'LAST`**
- En plus des opérateurs sur les types scalaires, **`Float`** possède des opérateurs arithmétiques :
  - opérateurs unaires : **`+`**, **`-`**, **`abs`**
  - opérateurs binaires : **`+`**, **`-`**, **`*`**, **`/`**, **`**`**, **`rem`**, **`mod`**
- Il existe d'autres types de flottants : **`Short_Float`** (nombre de chiffres significatifs plus réduit que **`Float`**) et **`Long_Float`** (nombre de chiffres significatifs plus grand que **`Float`**)
- L'attribut **`DIGITS`** donne le nombre maximum de chiffres significatifs pour les types à précision flottante
- L'attribut **`DELTA`** donne le nombre de chiffres significatifs pour les types à précision fixe

Float (2/2)

- Définition d'un nouveau type flottant en précisant le nombre maximum de chiffres significatifs (cette précision n'est pas obligatoire) :

```
type Reel is digits 6 range -10.00..+10.0;
```

- Cette définition équivaut à définir un type dérivé et un sous type :

```
type Temp is new Float;
subtype Reel is Temp digits 6 range -10.00..+10.0;
```

- Définition d'un nouveau type fixe en précisant la résolution (cette précision n'est pas obligatoire) :

```
type Reel is delta 0.01 range 0.0..+100_000.0;
```

- De nombreux attributs supplémentaires existent pour les types réels flottants ou fixe (valeurs possible du digit ou du delta, de la mantisse, de l'exposant, ...)

Priorité des opérateurs

- ADA gère la priorité des opérateurs et interdit les expressions ambiguës

- Récapitulatif des priorités des différents opérateurs :

opérateurs prioritaires	`** abs not`
opérateurs multiplicatifs	`* / rem mod`
opérateurs additifs unaires	`+ -`
opérateurs additifs binaires	`+ - &`
opérateurs de comparaison	`= /= <= >= < >`
opérateurs logiques	`and or xor`

Access (1/3)

- Les pointeurs sont des accès aux objets ou sous-programmes
- Un pointeur se définit à l'aide du type d'objet (ou de sous-programme) vers lequel il pointe

```
type Acces_sur_entier is access Integer;
```

- Les pointeurs en ADA obéissent à certaines règles pour éviter les erreurs :
  - un pointeur ne peut pointer que des objets d'un même type
  - un pointeur ne peut pointer sur un objet référencé par un identificateur sauf s'il est déclaré **`aliased`**
  - un pointeur a toujours une valeur qui est soit **`null`** soit définie par allocation (opérateur **`new`**)

Access (2/3)

```
access_type_definition ::= [null_exclusion] access_to_object_definition |
                           [null_exclusion] access_to_subprogram_definition

access_to_object_definition ::= access [general_access_modifier] subtype_indication

general_access_modifier ::= all | constant

access_to_subprogram_definition ::= access [protected] procedure parameter_profile |
                                    access [protected] function  parameter_and_result_profile

null_exclusion ::= not null

access_definition ::= [null_exclusion] access [constant] subtype_mark |
                      [null_exclusion] access [protected] procedure parameter_profile |
                      [null_exclusion] access [protected] function parameter_and_result_profile
```

- Deux modifieurs permettent de paramétrer le pointeur :
  - **`all`** indique que le pointeur peut donner un accès à des variables, qui peuvent être modifiées via le pointeur
  - **`constant`** indique que le pointeur ne peut donner accès qu'à des variables pointées (créées par allocation uniquement)
- On peut interdire qu'un pointeur ait la valeur **`null`**

Access (3/3)

- Allocateur : opérateur **new**

```
type Acces_sur_entier is access Integer;
ae : Acces_sur_entier;
ae = new Integer;
```

- On peut aussi préciser la valeur de la variable pointée à l'allocation par l'utilisation d'une expression qualifiée

```
ae = new Integer'(21);
```

- L'accès aux variables pointées se fait par l'opérateur **all**

```
titi : Integer;
titi := ae.all;
ae.all := 27;
```

- L'usage des pointeurs est toujours très délicat et il est préférable de s'en passer

Tableaux (1/2)

- Un type tableau est un type composé d'éléments tous du même type, indicés par un type discret.

- Un tableau a une taille fixée par un ensemble d'intervalles, mais qui peut être définie soit à la déclaration (tableau contraint) soit à la création (tableau non contraint).

array_type_definition ::= unconstrained_array_definition | constrained_array_definition

unconstrained_array_definition ::= **array (** index_subtype_definition { **,** index_subtype_definition } **)**
of component_definition

index_subtype_definition ::= subtype_mark **range <>**

constrained_array_definition ::= **array (** discrete_subtype_definition { **,** discrete_subtype_definition } **)**
of component_definition

discrete_subtype_definition ::= *discrete*_subtype_indication | range

component_definition ::= [**aliased**] subtype_indication | [**aliased**] access_definition

Tableaux (2/2)

- *Exemples de types tableaux contraints* :

```
type Table is array(1 .. 10) of Integer;

type Jour is (Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche);
type Cours is (Cours_Magistral, TD, TP, Seminaire);
type Agenda is array(Jour) of Cours;

type Matrice is array(1..10,1..20) of Float;
```

- *Exemples de types tableaux non contraints* :

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

- *Exemples d'utilisation d'un tableau non contraint* :

```
subtype Vecteur_3 is Vecteur(1..3);
subtype Matrice_22 is Matrice(1..2, 1..2);
subtype Vecteur_2 is Vecteur_3(1..2); -- interdit par le compilateur
subtype Matrice_2X is Matrice(1..2, range <>); -- interdit par le compilateur

v3 : Vecteur_3;
v3Bis : Vecteur(1..3);
v:Vecteur; -- interdit par le compilateur
```


Attributs des tableaux

- L'attribut FIRST(n) donne la borne inférieure du nième indice du tableau. L'attribut LAST(n) donne la borne supérieure du nième indice du tableau. L'attribut RANGE(n) donne l'intervalle du nième indice du tableau. L'attribut LENGTH(n) donne le nombre de valeurs comprises dans l'intervalle du nième indice du tableau

- *Exemple : le programme ci-dessous écrit 8 et 11 à l'écran*

```
procedure Test is

type Jour is (Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche);
type Activité is (Cours_Magistral, TD, TP, Seminaire, Travail_Personnel);
type Agenda is array(Jour,8..18) of Activité;
emploi_du_temps : Agenda;

begin
   emploi_du_temps(Lundi,8) := TD;
   emploi_du_temps(Lundi,11) := Seminaire;
   emploi_du_temps(Mardi,10) := Cours_Magistral;
   put(emploi_du_temps'FIRST(2));
   put(emploi_du_temps'LENGTH(2));
end;
```

Agrégats (1/2)

- Un agrégat de tableau peut permettre de définir les valeurs d'un tableau :

```
array_aggregate ::= positional_array_aggregate | named_array_aggregate

positional_array_aggregate ::= ( expression, expression { , expression } )
                             | ( expression { , expression } , others => expression )
                             | ( expression { , expression} , others => <> )

named_array_aggregate ::= ( array_component_association { , array_component_association } )

array_component_association ::= discrete_choice_list => expression | discrete_choice_list => <>
```

- Un agrégat doit vérifier les règles suivantes :
  - tout élément doit avoir une valeur et une seule
  - le choix `others` doit être le dernier
  - on ne peut combiner des associations de position et des associations par nom

Agrégats (2/2)

- *Exemples de définitions des valeurs de tableaux à l'aide d'agrégats* :

```
type IntTab is array(0..9) of Integer;
tab1 : IntTab := (1,4,4,1,1,1,5,0,5,3);
tab2 : IntTab := (0 | 3..5 => 1, 1..2 => 4, 6 | 8 => 5, 7 => 0, 9 => 3);
tab3 : IntTab := (1..2 => 4, 6 | 8 => 5, 7 => 0, 9 => 3, others => 1);
```

- Il est possible de déclarer un objet de type tableau au moyen d'un type anonyme :

```
type Jour is (Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche);
type Activité is (Cours_Magistral, TD, TP, Seminaire, Travail_Personnel);
emploi_du_temps : array(Jour,8..18) of Activité;
```

- On peut utiliser des tranches de tableaux unidimensionnels définies par un intervalle :

```
type Journée is array(8..18) of Activité;
j : Journée;
j(12..18) := (12..18 => Travail_Personnel);
```

Chaîne de caractères

- Une chaine de caractères est un tableau de caractères :

```
type String is array(Positive range <>) of Character;
```

- On peut définir un objet de type **String** par un agrégat ou utiliser la notation avec guillemets :

```
bonj : String := ('b','o','n','j','o','u','r');
aure : String := "au revoir";
```


Enregistrement (1/2)

- Un enregistrement est un objet composé dont les composants sont identifiés par des noms :

record_type_definition ::= [[**abstract**] **tagged**] [**limited**] record_definition

record_definition ::= **record**
component_list
end record
| **null record**

component_list ::= component_item { component_item } | { component_item } variant_part | **null ;**

component_item ::= component_declaration | aspect_clause

component_declaration ::= defining_identifier_list : component_definition [**:=** default_expression] ;

- *Exemple :*

```
-- on suppose qu'une type Nom_Mois existe
type Date is record
   Jour : Integer range 1 .. 31;
   Mois : Nom_Mois;
   Année : Integer range 0 .. 4000;
end record;
```

Enregistrement (2/2)

- Accès aux champs d'un enregistrement

```
d : Date;
d.Jour := 22; d.Mois := Mars; d.Année := 2007;
```

- Spécification des valeurs par défaut des champs d'enregistrement

```
-- on suppose qu'une type Nom_Mois existe
type Date is record
   Jour : Integer range 1 .. 31 := 22;
   Mois : Nom_Mois := Mars;
   Année : Integer range 0 .. 4000 := 2007;
end record;
```

- Les agrégats d'enregistrement sont similaires aux agrégats de tableaux

```
d : Date := (22,Mars,2007);
d : Date := (Jour => 22, Mois => Mars, Année => 2007);
```


Enregistrement à variant (1/2)

- Il est possible de définir des champs différents en fonction de paramètres

variant_part ::= **case** *discriminant*_direct_name **is** variant { variant } **end case ;**

variant ::= **when** discrete_choice_list **=>** component_list

discrete_choice_list ::= discrete_choice { | discrete_choice }

discrete_choice ::= expression | discrete_range | **others**

Enregistrement à variant (2/2)

- *Exemples* :

```
type Device is (Printer, Disk, Drum);
   type State  is (Open, Closed);
type Peripheral(Unit : Device := Disk) is record
       Status : State;
       case Unit is
           when Printer => Line_Count : Integer range 1 .. Page_Size;
           when others =>
               Cylinder   : Cylinder_Index;
               Track      : Track_Number;
       end case;
end record;

subtype Drum_Unit is Peripheral(Drum);
   subtype Disk_Unit is Peripheral(Disk);

Writer    : Peripheral(Unit  => Printer);
   Archive   : Disk_Unit;
```


Types à discriminant (1/2)

- Il est possible d'utiliser des discriminants pour définir des types dont tous les paramètres ne sont pas fixés à la déclaration du type (partie **`known_discriminant_part`** d'une déclaration)

```
discriminant_part ::= unknown_discriminant_part | known_discriminant_part

unknown_discriminant_part ::= (<>)

known_discriminant_part ::= ( discriminant_specification { ; discriminant_specification } )

discriminant_specification ::= defining_identifier_list : [ null_exclusion ] subtype_mark [ := default_expression ]
                             | defining_identifier_list : access_definition [ := default_expression ]

default_expression ::= expression
```

- Le discriminant indique une liste de paramètres avec leurs types et éventuellement leurs valeurs par défaut
- C'est un des aspects de la généricité en ADA

Types à discriminant (2/2)

- *Exemples* :

```
type Mat is array(Integer range <>,Integer range <>) of Float;
type Matrice(Lignes,Colonnes : Integer) is record
    nom:String(1..10);
    données:Mat(1..Lignes,1..Colonnes);
end record;

type Matrice_Carrée(Dim : Integer := 2) is new Matrice(Lignes => Dim,
   Colonnes => Dim);

m : Matrice(3,4);
c : Matrice_Carrée(6);
d : Matrice_Carrée;
```


Déclaration de type incomplet

- Une déclaration de type incomplet sert uniquement à réserver un mot-clé pour désigner un type qui sera obligatoirement défini plus loin. Elles servent surtout pour définir des pointeurs (**access**) sur des objets non encore définis.

```
déclaration_de_type_incomplet ::= type identificateur [partie_discriminants] ;
```

- *Exemple : définir une liste chainée en ADA nécessite d'utiliser un type incomplet*

```
type Element;
type Ptr_Element is access Element;
type Element is record
   suivant : Ptr_Element;
   valeur : Integer;
end record;
```