Introduction - Langage ADA

Objectifs de la séance :
- avoir une petite idée de la manière d'écrire un programme en Ada pour pouvoir se débrouiller en TP, certaines parties seront approfondies dans les cours suivants
- écrire quelques algorithmes pour redémarrer

Remarque : l'objectif premier du cours n'est pas d'apprendre un langage mais des notions d'algorithmique. Le langage choisi n'est qu'un support au cours d'algorithmique.

1 Historique - Généralités

1974 concours lancé par le département de la défense des Etats-Unis
1983 gagnant : langage ADA (Augusta Ada Byron comtesse de Lovelace (1815-1852), travaille avec Babbage, considérée comme première programmeuse de l'histoire), normalisé ANSI
1987 première norme ISO : Ada83
1995 révision, norme ISO Ada95 (évolution des conceptes de programmation : programmation orientée objet)
2007 nouvelle révision : Ada 2005
- offre : sécurité, lisibilité, portabilité, modularité
- permet de supporter plusieurs styles de programmation : fonctionnelle, procédurale, objet
- facilité d'interfaçage avec d'autre langages
- logiciels fiables
- informatique embarquée, systèmes temps réels
- conçu pour développer de très grosses applications
- domaines d'application :
  - militaire
  - avionique : airbus, boeing (777)
  - nucléaire : EDF
  - transport terrestre : TGV
  - espace : Ariane 4 et 5
  - communications

2 Caractéristiques

- langage impératif : élément de base = instruction, une instruction est composée d'expressions
- langage compilé : code source → compilateur → programme objet executable, ensuite execution du programme objet
- points forts :
  - typage fort : sécurité
  - paquetage : structuration architecturale (décomposition modulaire) : réutilisabilité, facilité de maintenance, facilité pour travail en équipe
  - Abstraction de données : types privés : réutilisabilité, sécurité
  - programmation orientée objet : types extensibles
  - gestion des erreurs à l'execution : exceptions
  - programmation paramètrée par le type des données : généricité
  - programmation parallèle : tâches
  - facilité d'interfaçage avec d'autres langages : C, C++ etc

3 Organisation générale d'un programme ADA

Un programme ADA s'articule autour d'un programme principal qui utilise le plus souvent des *paquetages* correspondant à des unités cohérentes :

- *regroupement par fonctionnalité :* entrées/sorties, bibliothèques mathématiques, ...
- *type abstrait :* nombres rationnels, polynômes, listes, ...
- etc.

3.1 Structure du programme principal

Le programme principal doit en général assurer l'interface avec l'utilisateur :

- avec des instructions d'entrée : l'utilisateur fournit au clavier les données sur lesquelles va travailler le programme
- avec des instructions de sortie : affichage à l'écran ou dans un fichier des résultats du programme
- en utilisant les arguments qui lui sont passés en ligne de commande
- en manipulant des fichiers

Il appel des procédures et des fonctions qui peuvent être dans d'autres fichiers source : sous-programmes ou paquetages.

Il se présente ainsi dans un fichier qui porte nécessairement le nom *nom_programme.adb* :

```
with nom_paquetage1;
use nom_paquetage1;
procedure nom_programme is
   partie déclarative
begin
   corps du programme
end nom_programme;
```

La clause `with` permet d'accéder aux fonctionnalités du paquetage `nom_paquetage1`. La clause `use` est ensuite facultative, mais bien pratique : elle permet d'invoquer les fonctionnalités de `nom_paquetage1` directement, sans préfixer l'appel de `nom_paquetage1.` On peut l'utiliser tant qu'il n'y a pas de risque d'ambiguité avec un autre paquetage. Des exemples concrèts d'utilisation de `use` et `with` seront présentés dans la section 8 sur les paquetages d'entrée-sortie par exemple.

Le programme le plus simple comporte les quatre lignes suivantes :

```
procedure fait_rien is
begin
      null ;
end fait_rien;
```

Il faut au moins une instruction entre **begin** et **end**, l'instruction **null** ne fait rien du tout, mais c'est tout de même une instruction. Elle peut être utilisée dans certaines structures conditionnelles pour indiquer qu'un cas n'a pas été oublié par le programmeur mais qu'aucune action n'est nécessaire pour son traitement.

3.2 Compilation

La commande que nous utiliserons en TP pour compiler les programmes est `gnatmake`. Afin de pouvoir l'utiliser vous devrez modifier la variable `PATH` de votre environnement soit en début de chaque TP par l'execution de la commande

```
PATH=/opt/gnat/bin:$PATH
```

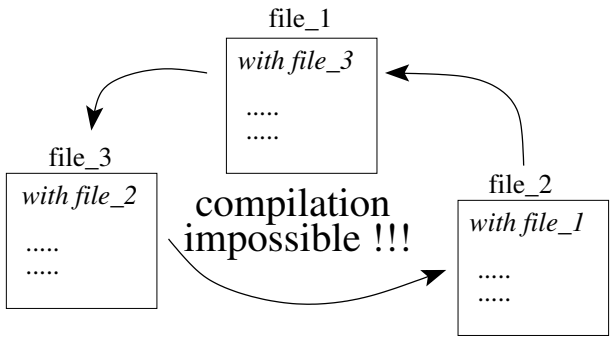
soit une fois pour toutes en incluant la commande précédente dans votre fichier `.bashrc` associé au lancement de votre shell.

Ensuite l'execution de la commande :

```
gnatmake programme_principal;
```

provoque la (re)compilation de toutes les unités de compilation (paquetages et sous-programmes) qui doivent l'être pour créer le binaire executable correspondant à votre programme principal.

Attention aux dépendances entre les unités de compilation, il ne doit pas y avoir de cycle sinon la compilation impossible comme dans l'exemple suivant.



3.3 Structure d'un paquetage

Un paquetage est une sorte de boîte qui fournit un ensemble de fonctionnalités. Un paquetage se partage en deux fichiers :

- `nom_paquetage.ads` qui contient la **spécification** du paquetage (fonctionnalités visibles) : énumération des types, fonctions et procédures proposées par le paquetage

  ```
  package nom_paquetage is
     déclarations des types, fonctions, procédures...
  end nom_paquetage;
  ```

- `nom_paquetage.adb` qui contient l'implémentation du paquetage. Cette partie peut également définir des procédures ou des fonctions cachées à l'utilisateur.

  ```
  package body nom_paquetage is
     définitions des fonctions, procédures...
  end nom_paquetage;
  ```

Lorsqu'un paquetage `pack_1` utilise un autre paquetage `pack_2` c'est dans le fichier `pack_1.adb` que doit se trouver la clause `with pack_2`, et éventuellement aussi `use pack_2`.

4 Les types prédéfinis

En Ada chaque "objet" ou expression a un type unique, la cohérence des types dans les expressions est vérifiée statiquement à la compilation.

- Le type entier `INTEGER`. Les opérations courantes prédéfinies sont `+`, `*`, `-`, `/` (division euclidienne), `mod` (reste de la division) , `**` (exponentiation) etc.
- Le type caractère `CHARACTER`. Un caractère constant est précisé entre apostrophes : `'Z'`.
- Le type booléen `BOOLEAN`, composé de `TRUE` et `FALSE`. Les connecteurs sont `and`, `or`, `not`, `or else`, `and then`. Le connecteur `or else` a la même sémantique que `or`, mais son mode d'évaluation est différent : si le premier argument est vrai, le second n'est pas évalué. De même, `and then` a la même sémantique que `and`, mais le second argument n'est évalué que si le premier est vrai.
- le type `FLOAT` pour les nombres réels,
- le type `STRING` pour les chaînes de caractères est détaillé en section 12. Une chaîne de caractères constante est précisée entre guillemets : `"Zorro"`

Pour chaque type il existe ou peut exister des sous-types, ou ensembles restreints de valeurs du type. Par exemple Integer a trois sous-types : Integer, Natural (≥ 0) et Positive (> 0). L'appartenance à un sous-type est vérifié à l'execution (dynamiquement). Par exemple si on définit la fonction :

function bete(x : Positive) **return** Positive **is**
begin
    return x ;
end bete;

les erreurs suivantes vont être détéctées soit à la compilation, soit à l'execution :

- y : Positive ;
- y := (n=1) : erreur détéctée à la compilation : on essaie d'affecter un booléen à une variable de type Positive
- y := bete(n-3) : OK compilation car bete renvoie bien un Positive comme y, mais erreur à l'execution si n-3 ≤ 0

Les types `INTEGER` (et ses sous-types), `CHARACTER` et `BOOLEAN` ont en commun d'être des *types discrets* : les valeurs sont totalement ordonnées, avec un prédecesseur et un succésseur. Par exemple, 2 est le successeur de 1 et le prédecesseur de 3. De même, `'c'` est le successeur de `'b'` et le prédecesseur de `'d'`.

Nous verrons par la suite que ces types bénéficient d'attributs spécifiques qui facilitent leur manipulation, par exemple :

- character'Pos('a') $\rightarrow$ 97 : position du caractère 'a' dans la définition du type
- character'Val(65) $\rightarrow$ 'A' : caractère à la position 65

5 La partie déclarative

Une partie déclarative apparaît de manière naturelle dans le programme principal (définition globale), ainsi qu'avant chaque corps de fonction et procédure (définition locale). Elle contient la déclaration des types et sous-types, détaillée à la section 10, des constantes, des variables et des procédures et fonctions utilisées ensuite dans le corps.

5.1 Identificateurs et mots réservés

Les identificateurs (noms des variables, constantes, fonctions et procédures) doivent commencer par une lettre et peuvent comporter des lettres minuscules et majuscules, des chiffres, le caractère _ souligné.

De plus il existe des mots réservés qui ne peuvent pas être utilisés comme identificateurs, il s'agit des mots qui structurent le langage (procedure, function, begin, end etc).

5.2 Variables et constantes

variable <nom> : <type> ;
- x : Integer ; y : Integer ; z : Integer ;
- $\rightarrow$ x,y,z : Integer ;

variable initialisée <nom> : <type> := expression ;
- x : Integer := 3 ; y : Integer := 4 ;
- z : Integer := x+y-12 ;
- u : Integer := z+3 ; v : Integer := z+3 ;
- $\rightarrow$ u,v : Integer := z+3 ;

constante <nom> : **constant** <type> := expression ;
- x : Integer := 3 ;
- A,B : **constant** Integer := x*2 ;
- $\rightarrow$ A et B sont des constantes entières égales à 6

5.3 Les fonctions

```
function Nom_fonction(arg1:type1;arg2:type 2...) return type_resultat is
   partie déclarative
begin
   instructions
   return ...;
end Nom_fonction;
```

Les arguments d'une fonction sont accessibles en lecture uniquement. En particulier, il est hors de question de modifier leur valeur lors de l'exécution de la fonction.

Une fonction retourne un résultat et doit se terminer par l'instruction **return**. Le type de l'objet retourné doit être identique à celui annoncé dans la signature de la fonction, l'execution s'arrête à la première instruction **return** rencontrée.

Voici par exemple la fonction carré :

function carré(x : Integer) **return** Integer **is**
begin
    return x*x ;-- x**2 marche aussi
    x := 2 ;-- ne sera jamais executé
end carré;

Dans l'idée, une fonction ada se comporte comme une fonction mathématique, il faut considérer un appel à une fonction comme un objet du type qu'elle retourne :

INCORRECT			CORRECT
...		...	...
begin		**begin**	x : Integer ;
...		...	**begin**
carré(3) ;	$\Leftrightarrow$	9 ;	...
...		...	x := carré(4) ;
end ;		**end** ;	**return** carré(3)+x ;
			...
			end ;

Dans la spécification d'un paquetage (le fichier .ads), si l'on souhaite qu'une fonction soit visible par un utilisateur, l'information qui doit apparaître est

```
function Nom_fonction(arg1:type1; arg2:type 2...) return type_resultat;
```

5.4 Les procédures

Une procédure ne retourne pas de résultat au même sens qu'une fonction. Elle ne contient pas d'instruction return, mais par le biais de ses modes de passage de paramètres peut quand même transmettre des données au programme qui l'appel.

```
procedure Nom_procedure(arg1:mode1 type1; arg2:mode2 type 2...) is
   partie déclarative
begin
   instructions
end Nom_procedure;
```

Pour chaque paramètre de la procédure, il faut donc préciser la modalité de passage :

- `in` : paramètre donnée (passage par valeur) : accès en lecture uniquement, comme l'argument d'une fonction, valeur constante : on ne sait pas où elle est rangée en mémoire pour pouvoir aller la modifier.

- `out` : paramètre résultat (passage par adresse), accès en écriture uniquement. Le paramètre est équivallent à une variable non initialisée, la procédure ne doit pas essayer de l'utiliser avant de l'avoir initialisé, elle doit lui affecter une valeur avant la fin de son execution. Le paramètre correspond à un résultat de sortie, c'est un emplacement mémoire auquel le programme sait accèder. Cet emplacement est vide au départ et la procédure doit lui affecter une valeur. Le paramètre effectif, c'est-à-dire celui qui est donné à la procédure lors d'un appel doit donc être un objet modifiable, par exemple pas une constante entière comme 3. C'est une boîte vide à remplir.
- `in out` : paramètre donnée/résultat (passage par adresse), le mode le plus souple, qui autorise de prendre en compte et de modifier la valeur du paramètre. Equivallent à une variable initialisée, sa valeur initiale est celle du paramètre effectif au moment de l'appel, une nouvelle valeur peut être attibuée au paramètre effectif au cours de la procédure, après l'execution la valeur du paramètre est la dernière attribuée avant la fin de la procédure. Emplacement mémoire connu contenant déjà une valeur que l'on peut utiliser et modifier, boîte pleine dont on peut utiliser le contenu et le remplacer.

Contrairement à un appel de fonction qui est considérée comme une expression, un appel de procédure doit être considérée comme une instruction à part entière :

EXEMPLE	INCORRECT	CORRECT

procedure fait_rien **is**
begin
  null ;
end fait_rien;
	...
	  x : type_quelque_chose ;
	begin
	  ...
	  x := fait_rien ;
	  ...
	end ;
		...
		begin
		  ...
		  fait_rien
		  ;
		  ...
		end ;

Remarque : on peut donner une valeur par défaut aux paramètres d'une procédure, c'est la valeur par défaut qui est utilisée comme paramètre effectif si aucun n'est donné au moment de l'appel.

Dans la spécification d'un paquetage, l'information qui doit apparaître est

```
procedure Nom_procedure(arg1:mode1 type1; arg2:mode2 type 2...);
```

5.5 Surchage

Deux identificateurs de procédure ou fonction sont surchargeables si les profils différent au moins par l'un des éléments suivants :
- nombre de paramètres
- type d'au moins un paramètre
- type du résultat

Les paquetages d'entrée/sortie (section 8) donnent beaucoup d'exemple de surcharge.

6 Structure de bloc et portée des déclarations

Maintenant qu'on sait déclarer des variables, constantes, procédures et fonctions, quelle est la durée de vie de ces déclarations ?

6.0.1 Bloc

Un *bloc* est une région de programme introduite par une déclaration de sous-programme. Elle est constituée de :
- déclaration des paramètres formels
- déclarations locales du sous-programme

– instructions du corps du sous-programme

Un identificateur (variable, constante, fonction ou procedure) doit toujours être déclaré dans un bloc.

En bleu le bloc introduit par la déclaration du sous-programme `titi` :

```
procedure titi(<paramètres>) is
        <déclarations> ;
begin
        <instructions> ;
end titi;
```

6.0.2 Portée d'une déclaration

Un identificateur 'existe' et peut être utilisé de la fin de sa déclaration jusqu'à la fin du bloc où il est déclaré. Dans l'exemple suivant `x` est déclaré dans le bloc bleu, mais il n'existe qu'à partir de sa définition (zone en gras), on ne peut pas s'en servir pour initialiser `y`, mais pour `z` il n'y a pas de problème :

```
procedure titi(<paramètres>) is
        y : constant Integer := 3 ;
        x : Integer := 4 ;
        z : constant Integer := x ;
        <déclarations> ;
begin
        <instructions> ;
end titi;
```

6.0.3 Visibilité d'un identificateur

Un identificateur peut être masqué par un autre à l'intérieur de sa zone de portée :

```
with Ada.Integer_Text_IO ;
use Ada.Integer_Text_IO ;
procedure P is
        x : Integer ;
        procedure Q is
        begin
                put(x) ;
        end Q;
        procedure R is
                x : Integer ;
        begin
                x := 12 ;
                Q ;
                put(x) ;
        end R;
begin
        x := 45 ;
        R ;
        put(x) ;
end P;
```

Le `x` déclaré dans `R` masque le `x` déclaré dans `P`, à l'execution les affichages sont les suivants :

– 45 → `Put(x)` dans `Q` appelé par `R` appelé par `P` : le `x` vu par `Q` est bien celui déclaré dans `P`

– 12 → `Put(x)` dans `R` appelé par `P` : le `x` vu dans `R` est celui déclaré dans `R`

– 45 → `Put(x)` dans `P`, le `x` déclaré dans `R` n'existe pas dans le corps de `P` car on est en dehors du bloc introduit par la définition de `R`

Remarque : deux identificateurs identiques ne peuvent pas être déclarés dans le même bloc sauf s'ils sont surchargeables : identificateurs de sous-programme (procédure ou fonction) ayant des profils différents.

6.0.4 Déclaration locale à un bloc

Il est possible d'ajouter une partie déclarative avant tout bloc **begin** ...**end** à l'interieur d'un programme, à l'aide du mot clef **declare**. Les déclarations ainsi faites ne seront alors **uniquement définies pour ce bloc** (*déclaration locale au bloc*).

```
...
declare
      <déclarations> ;
begin
      <instructions> ;
end ;
...
```

7 Structures algorithmiques

7.1 Structures conditionnelles

Il y a un *If then else* classique ainsi qu'un second avec des conditions imbriquées. La partie *else* est facultative pour les deux structures.

```
if condition then
   ...
else
   ...
end if;
```

```
if condition then
   ...
elsif condition2 then
   ...
elsif condition3 then
   ...
else    ...
end if;
```

Il est également possible de faire des choix multiples.

```
case expression is
   when choix1 => ...
   when choix2 => ...
   ...
   when others => ...
end case;
```

La variable *expression* est de type discret : `Integer`, `Boolean`, `Character`. L'énumération des cas doit être **exhaustive** (couvrir toutes les possibilités). Cela explique qu'il faut souvent avoir recours au cas par défaut `others`, éventuellement avec l'instruction `null`.

Un petit exemple de liste de valeurs 1|3..4|8..10|14|20..23 pour les choix de l'instruction **case**.

7.2 Structures itératives

pour	tant que	loop
for <indice> **in** <intervalle> **loop** <instructions> ; **end loop**; **for** <indice> **in reverse** <intervalle> **loop** <instructions> ; **end loop**;	**while** <condition> **loop** <instructions> ; **end loop**;	**loop** <instructions> ; **end loop**; On utilise l'instruction **exit** pour sortir de la boucle.

Boucle pour

- L'<indice> utilisé ne doit pas être déclarer à l'avance mais il n'est visible qu'à l'intérieur de la boucle et ne peut être modifié pas des instructions explicites (variable muette). Cet indice est de type discret comme `INTEGER` ou `CHARACTER`.

- L'<intervalle> décrit l'intervalle de variation de l'indice de la boucle. Il est de la forme `debut .. fin` où `debut` et `fin` sont des valeurs discrètes.

- Exemple : `2..5`. Si on veut que le compteur de boucle décroit de, disons, 5 à 2, il faut préciser que l'intervalle doit être parcouru en sens inverse : `reverse 2..5` (et non `5..2`, qui est un intervalle vide).

- On peut aussi parcourir des intervalles de caractères ou d'autres types énumérés en précisant de quel type il s'agit, la syntaxe de la boucle pour est alors : `for i in character range 'a'..'z' loop ...`.

8 Entrées-sorties

Pour afficher à l'écran ou fournir à un programme des informations par l'intermédiaire du clavier nous utiliserons les deux paquetages d'entrées-sorties suivants :

- *Ada.Integer_ Text_ IO* : Input/Output pour les entiers
- *Ada.Text_ IO* : Input/Ouput pour les chaînes de caractères

Pour les utiliser deux instructions à placer en tout début du fichier.abd où on souhaite s'en servir sont nécessaires :

with <nom_du_paquetage> ; placé en début du fichier contenant un programme, permet d'utiliser les fonctionnalités offerte par le paquetage, par exemple pour afficher un entier on utilise l'instruction `Ada.Integer_Text_IO.put(<entier>);`

use <nom_du_paquetage> ; placé en début du fichier contenant un programme, permet d'abréger les notations, pour afficher un entier on peut alors utiliser : `put(<entier>);`

Ces paquetages fournissent entre autre les procédures suivantes :

procedure get(X : out Integer) procédure qui lit un entier au clavier et met sa valeur dans la variable X

procedure put(X : in Integer) procédure qui affiche la valeur de la variable X

procedure get(X : out Character) procédure qui lit un caractère au clavier et le met dans la variable X

procedure put(X : in Character) procédure qui affiche le caractère contenu dans la variable X

procedure put(X : in Integer, T : in Integer) procédure qui affiche la valeur de la variable X sur le nombre de caractères donné par T. Si le nombre de caractères donnés est

insufisant, il est remplacé par le nombre minimum nécessaire.

procedure put_line(S : in String) affichage et saut de ligne (uniquement pour du texte)

procedure new_line changement de ligne

Quelque éléments permettent de faciliter l'affichage mélangé de chaînes de caractères et de résultats entiers :

- L'opérateur `&` effectue la concaténation de chaînes de caractères.
- `Integer'Image(<valeur de type Integer>)` est la chaîne de caractère qui représente `<valeur de type Integer>` : `Integer'Image(123)` est la chaîne " 123" (avec un espace devant si positif et '-' sinon).
- `Integer'Value(<chaîne représentant un entier>)` est la valeur de l'entier représenté par la chaîne `<chaîne représentant un entier>` : `Integer'Value("123")` vaut 123.

Un petit exemple utilisant ces attributs et opérateurs :

```
procedure calcul is
        x : Integer ;
begin
        put("Entrer un entier : ") ;
        get(x) ;
        put("Le carré de cette valeur est :") ;
        put(carre(x)) ;
        New_Line ;
end calcul;
```

```
procedure calcul is
        x : Integer ;
begin
        put("Entrer un entier : ") ;
        get(x) ;
        put("Le carré de " & Integer'Image(x)
        & "est : " & Integer'Image(carre(x))) ;
        New_Line ;
end calcul;
```

9 Arguments d'un programme

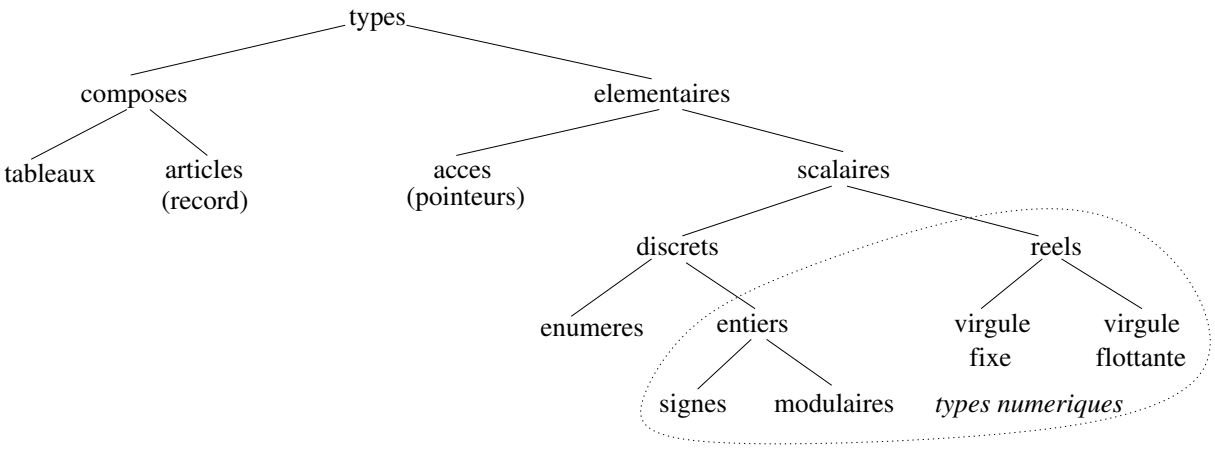
Le paquetage Ada.Command_Line permet de récuperer des arguments passés aux programme en ligne de commande. Il contient en particulier les trois fonctions suivantes :

- function **Argument_Count** return Natural : retourne le nombre d'arguments passés au programme, 0 s'il n'y en a aucun.
- function **Argument**(N : Natural) return String : retourne la chaîne de caractères correspondant à l'argument en position N dans la liste des arguments. Dans cette liste les arguments sont numérotés de 1 à Argument_Count.
- function **Command_Name** return String : retourne la chaîne de caractères correspondant au nom du programme en cours d'execution.

10 Types

10.1 Organisation des types

En ada un type est définit par un ensemble de valeurs et un ensemble d'opérations. Ils sont regroupés en catégories selon la hiérarchie suivante :



L'appartenance à un type est vérifiée statiquement (lors de la compilation).

Il existe des types prédéfinis mais il est aussi possible d'en définir de nouveaux :

- préciser un type à partir d'un type existant (sous-type)
- construire un type de toute piéce : record, tableau, enumerate
- à partir d'un type modèle (type dérivé)

Par exemple le type COMPLEXE n'existe pas, mais on peut définir :

- une structure avec partie réelle et imaginaire
- des opérations (surcharge des opérateurs possible)
- les relations avec les réels

On peut mettre tout ça dans un paquetage pour pouvoir manipuler des objets complexes aussi simplement que des objets entiers.

```
package COMPLEXE is -- fichier COMPLEXE.ads
      ...
end COMPLEXE;
package body COMPLEXE is -- fichier COMPLEXE.adb
      ...
end COMPLEXE;
```

10.2 Sous-types

Un sous-type définit un sous-ensemble de valeurs d'un type parent existant déterminé par une contrainte.

L'appartenance à un sous-type ne peut être vérifiée qu'à l'execution du programme. La définition se fait de la manière suivante :

subtype <*nom sous-type*> **is** <*nom type parent*> <*contrainte*> ;

La forme de la contrainte dépend de la catégorie du type à partir duquel le sous-type est contruit.

Pour un type scalaire la contrainte est un intervalle :

```
subtype chiffre is Integer range 0..9 ;
subtype minuscule is Character range 'a'..'z' ;
subtype majuscule is Character range 'A'..'Z' ;
```

Les éléments d'un sous-type sont compatibles avec les éléments du type parent et les fonctions et opérateurs du type parent sont utilisables avec le sous-type, à condition bien sur de respecter la contrainte de sous-type :

```
j : Integer := 3 ;
i : chiffre ;
i := j ;-- OK à la compilation + à l'execution car j vaut 3 qui est bien un chiffre
i := i+4 ;-- toujours OK, i vaut 7
```

i := i*2 ;-- constraint_error levée à l'execution, la valeur de i sort de l'intervalle définissant le sous-type chiffre

De plus la contrainte est optionnelle : un sous type peut très bien être la copie de son type parent :

subtype mes_entiers **is** Integer ;

10.3 Types dérivés

Les types dérivés permettent de construire des arborescences de types convertibles entre eux (ils appartiennent à la même catégorie). Par exemple on peut définir les types vistesse, distance et accelération, qui pour des raisons de sécurité doivent être complétement distincts, mais qui peuvent intervenir dans un même calcul grâce à des conversions explicites, donc maîtrisées, de types.

Les valeurs d'un type dérivé sont les mêmes que celles de son type parent avec une contrainte supplémentaire optionnelle. Les opérateurs sont également hérités du type parent.

type <nom type dérivé> **is new** <type parent> <contrainte> ;

10.4 Conversions

Comme déjà évoqué, disposer de types distincts pour des valeurs identiques représentant des objets de nature différentes (carottes, navets etc) représente un atout important pour développer des applications sures. Cependant, pour effectuer des calculs par exemple, il est heureusement possible de convertir des éléments d'un type vers un autre dans les cas suivants :

1. entre deux types numériques, avec risque de perte de précision (ex réel vers entier)
2. entre deux types d'une même catégorie, car ils ont la même représentation en mémoire (deux sous-types ou deux types dérivés d'un même type)

→ nul part ailleur !

En supposant que l'on ait défini les types masse et volume comme deux types dérivés du type prédéfini Integer les lignes suivantes sont correctes à la compilation :

densité : constant := 10 ;
m : masse ;
v : volume ;
m := masse(v*densité) ;

Remarque : il existe tout de même des moyens pour effectuer une conversion entre deux types incompatibles selon les critères cités ci-dessus mais ces types de conversion ne sont à utiliser qu'en dernier recours.

11 Types scalaires

Les types scalaires regroupent à la fois les types discrèts et les types réels. Les valeurs des types scalaires sont totalement ordonnées. Pour tous ces types on dispose des comparateurs classiques : =, / = pour $\neq$, <, >, <= pour $\leq$ et >= pour $\geq$.

11.1 Attributs communs aux types scalaires

Les attributs sont des entités prédéfinies, intrinsèque au types, qui peuvent être des valeurs ou des fonctions.

- <nom_type>'First : première valeur du type (la plus petite), ex Natural'First = 0
- <nom_type>'Last :
- <nom_type>'Range :
- <nom_type>'Succ(v) :

– <nom_type>'Pred(v) :
– <nom_type>'Image(v) :
– <nom_type>'Value(chaîne) :

Certains attributs sont sensibles à la définition de sous-type mais les autres se comportent comme s'ils étaient appliqués au type de base, par exemple :

```
subtype Alphabet is Character range 'a'..'z' ;
Alphabet'First = 'a' ; -- OK
Alphabet'Last = 'z' ; -- OK
Alphabet'Succ('z') = '' ; -- 'z' n'a pourtant pas de successeur dans le sous-type Alphabet !
```

11.2 Types entiers (discret)

11.2.1 Entiers Signés

Il existe trois sous-types prédéfinis : Integer, Natural et Positive. On peut définir de nouveaux types :

type <nom_type>**is range** $v_1..v_2$;

Il faut que v_1 et v_2 soient des expressions entières calculables statiquement par le compilateur.

11.2.2 Entiers modulaires

Il s'agit des valeurs comprise entre 0 et m, où m est à définir mais doit être une expression entière calculable statiquement par le compialeur. Il n'y a aucun type modulaire prédéfini en ada.

type <nom_type>**is mod** m ;

Les régles de l'arithmétique modulaire sont particulières :

```
type octet is mod 256 ;
X : octet := 250 ;
...
X := X+9 ;-- ici X = 3 = 250 + 9 modulo 256
...
octet'First = 0 ;
octet'Last = 255 ;
octet'Succ(255) = 0 = 256 modulo 256 ;
```

11.2.3 Opérations définies sur les types entiers

Les opérations classiques sont définies sur les types entiers : +, - (unaire et binaire), * (produit), / (division entière), rem (reste de la division entière), mod (modulo), abs (valeur absolue), ** (exponentiation).

La différence entre A rem B et A mod B est la suivante :
– A rem B a le signe de A
– A mod B a le signe de B

Les opérateurs * et + ont les mêmes régles de priorité qu'en mathématique : 3+4*7 = 3+(4*7) = 31.

11.2.4 Littéraux entiers

Les constantes explicites comme 3, -8, 4 etc sont de tous les types entiers

11.3 Types énumérés (discret)

Il existe deux types énumérés prédéfinis : Boolean et Character, mais on peut en définir d'autres :

type <nom_type>**is** (<idf_1>, <idf_2>, ... , <idf_n>) ;

type couleur **is** (vert, orange, rouge) ;

couleur'First = vert ;

couleur'Range = vert..rouge ;

Pour les régles de surcharge, les identificateurs des valeurs sont considérées comme des fonctions sans paramètre, par exemple :

function vert **return** couleur;

Pour les types énumérés on dispose des deux attributs :

- <nom_type>'Pos(v) : position de la valeur v dans le type
- <nom_type>'Val(n) : valeur du type à la position donnée par n (Natural ?)

Les positions sont déterminées lors de la définition du type, la première est toujours 0 : couleur'Pos(vert) = 0 et couleur'Val(1) = orange.

11.4 Attribut supplémentaire pour les types discrèts

<nom_type>'Width : entier = longueur de la plus longue chaîne de caractères données par <nom_type>'Image(v) pour v de type <nom_type>, cet attribut est util pour la mise en page.

11.5 Types réels

Les types réels ne fournissent qu'une approximation des valeurs réelles, il y a par conséquent un manque de précision dans certains cas. Il existe deux catégories de types réels :

- point flottant : fournissent une erreur relative
- point fixe : fournissent une erreur absolue

Les littéraux réels sont composés d'une partie entière et d'une partie fractionnaire séparées par une virgule. Chaque partie est nécessaire. On peut aussi utiliser la notation scientifique :

- 4 → manque la partie fractionnaire et le point
- 4. → manque la partie fractionnaire
- 4.0 → OK
- .5 → manque la partie entière !
- 4.6e5 → OK : 4.6×10^5
- 5.23e-10 → OK

Les opérateurs classiques sont définies sur les types réels : +, - (unaire et binaire), * (produit), / (division réelle), abs (valeur absolue), ** (exponentiation), <, >, <=, >=, =, /=.

11.5.1 Point flottant

On doit indiquer le nombre **minimal** de chiffres décimaux significatifs après la virgule :

type MesReels **is digits** 7 ;

La précision est garantie sur au moins 7 chiffres après la virgule. Il existe deux types prédéfinis en Ada :

- Float : au moins 6 chiffres garantis
- Long_Float : au moins 15 chiffres garantis

L'attribut <nom_type>'Digits donne le nombre de chiffres significatifs après la virgule.

D'autres attributs donnent des informations sur la représentation des points flottants en mémoire : signe×mantisse×base$^{\text{exposant}}$.

- <nom_type>'machine_radix : base
- <nom_type>'machine_mantissa : nombre maximum de symboles de la mantisse
- <nom_type>'machine_emin et <nom_type>'machine_emax : définissent l'intervalle auquel appartient l'exposant

On peut également ajouter une contrainte d'intervalle :

type AutresReels **is digits** 5 **range** -0.4..23.34535 ;

11.5.2 Point fixe

Il y a deux catégories de points fixes :
- point fixe ordinaire (ordinary fixed point)
- point fixe décimaux (decimal fixed point)

Dans les deux cas l'idée est et de définir l'écart maximum **delta** entre 2 valeurs consécutives ainsi qu'un intervalle u..v.

Pour les points fixes ordinaires le delta est quelconque, donc attention car on ne peut pas forcément représenter tous les réels de la forme u+n×delta. Les éléments d'un type point fixe ordinaire n'ont pas tous la forme u+n×delta : on sait juste que la différence entre deux éléments consécutifs est inférieure à delta (puissance de 2 plus proche inférieure à delta).

type PointFixeOrdinaire **is delta** 0.0188 **range** -0.4..23.34 ;

Les points fixes décimaux ont la particularité que delta est une puissance de 10. On peut alors spécifier le nombre de chiffres significatifs **total**. Le delta entre les éléments du type sera exactement celui spécifié. Par exemple les éléments du type :

type PointFixeDecimal **is delta** 0.1 **digits** 3 **range** 0.0..10.0 ;

serons les réels : 0.0, 0.1, 0.2 etc 9.7, 9.8, 9.9 et 10.0. Attention, si un intervalle est donné, les bornes doivent bien respecter la restriction sur le nombre de chiffres (digits) indiquée, par exemple on ne peux pas définir un type sur 3 digits et mettre comme borne 100.0 car il y 4 digits ... La contrainte d'intervalle est optionnelle.

On peut utiliser l'attribut <nom_type>'Delta pour connaître le delta d'un type point fixe <nom_type>.

11.6 Utilisation des sous-types de types scalaires

Les sous-types peuvent être utilisés comme des ensembles dans des expressions booléennes, comme intervalle d'une boucle, ou dans une instruction case :

```
procedure Exemple is
        subtype Chiffre is Integer range 0..9 ;
        subtype Majuscule is Character range 'A'..'Z' ;
        subtype Minuscule is Character range 'a'..'z' ;
        X : Character ;
        Y : Integer ;
begin
        loop
                get(X) ;
                case X is
                        when Majuscule ⇒
                                put_line("c'est une majuscule") ;
                        when Minuscule ⇒
                                put_line("c'est une minuscule") ;
                        when others ⇒
                                put_line("c'est autre chose") ;
                                exit ;
                end case ;
        end loop;get(Y)
        if Y in Chiffre then
                put_line("C'est un chiffre") ;
        end if;
        for I in Chiffre loop
                put_line(Chiffre'Image(I**I)&" ") ;
        end loop;
```

end Exemple;

12 Tableaux et Chaînes de caractères

Un tableau est un objet composé, où tous les éléments sont de même type et sont accessibles directement par un indice. En ADA, il y a deux sortes de type tableau : les tableaux *contraints* et les tableaux *non contraints*.

12.1 Les Tableaux contraints

La taille du tableau est précisée lors de la définition du type :

```
type Nom_Type_Tableau is array(Intervalle_fini) of Type_Element;
```

où `Intervalle_fini` est un intervalle fini d'un type discret :

```
type T is array (1..5) of INTEGER ;
type MATRICE is array(1..4, 1..4) of INTEGER;
type C is array ('b'..'g') of BOOLEAN;
subtype DOUZAINE is POSITIVE range 1..12;
type TAB is array(DOUZAINE) of NATURAL;
type TABLE_DE_VERITE_A_DEUX_VARIABLES is array(BOOLEAN, BOOLEAN) of BOOLEAN ;
```

Quand on déclare un objet, il n'y a rien à préciser.

```
tab1 : T;
```

Les bornes du tableau ne sont pas nécessairement statiques :

```
procedure X(N : in NATURAL) ;
   type T is array (1..N) of INTEGER ;
begin
   ...
end X;
```

12.2 Les Tableaux non contraints

La définition du type précise le type des indices, le types des éléments, mais pas la taille du tableau.

```
type T is array (INTEGER range <>) of Natural ;
 -- un tableau de nombres naturels indexe par des entiers
```

Un objet de type `T` doit alors être contraint lors de sa déclaration, soit en précisant l'intervalle d'indexation, soit en lui affectant un autre tableau.

```
tab1 : T(3..8);
tab2 : T := tab1;     -- mêmes indices que tab1
tab3 : T(1..6):=T     -- indexe de 1 à 6
```

Attention ! Un tableau de type non-contraint ne peut pas être utilisé dans un enregistrement. Sinon, comment pourrait-on préciser la contrainte lors de la déclaration d'un objet ?

Une fois déclarés, les objets de type tableau, contraint ou non contraint, se manipulent tous de la même manière. Comment accède-t-on aux éléments d'un tableau ? `tab1(i)`.

12.3 Les attributs de tableaux

En ADA, les tableaux bénéficient d'attributs spécifiques, qui permettent de les manipuler en toute facilité. Ces attributs s'appliquent aussi bien aux tableaux contraints que non contraints.

`T` désigne un objet de type tableau.

`T'FIRST` indice du premier élément. Pour avoir la première valeur : `T(T'FIRST)`.

`T'FIRST (n)` indice du premier élément de la dimension n

`T'LAST` indice du dernier élément de la première dimension.

`T'LAST(n)` indice du dernier élément de la dimension n

`T'RANGE` intervalle d'indexation de la première dimension de `T`

`T'RANGE(n)` intervalle d'indexation de la dimension n de `T`

`T'LENGTH` longueur de la première dimension du tableau

`T'LENGTH(n)` longueur de la dimension n du tableau

Exemple : tester si une matrice est carrée.

```
type Matrice is array(Natural range <>, Natural range <>) of Natural;

function Est_Carre(m: Matrice) return Boolean is
begin
   return M'Length(1)=M'Length(2);
end Est_Carre;
```

12.4 Les agrégats

Il est possible de manipuler les cases d'un tableau à travers la notation d'*agrégat*.

```
  T:Tableau(1..6):=(1|2=>3, others=>4);
  U:Tableau:=(1..10=>3);
  V:Tableau:=(1..10=>3, others=>0);  -- non autorise
                                     -- "others" choice not allowed here
  W:Tableau(1..6):=(W'first=>0, others=>2); -- non autorise
                                            -- premature usage of "W"
  X:Tableau(1..20):=(1..10=>3); -- non autorise
           -- warning: too few elemnts for subtype of "Tableau" defined at line 107
           -- warning: "Constraint_error" will be raised at runtime
```

12.5 Les tranches

Un sous-tableau d'un tableau est encore un tableau et on peut y accéder directement en précisant les bornes des indices.

Exemple : récupérer les cinq derniers éléments sous forme de tableau.

```
function cinq_derniers(T: Tableau) return Tableau is
   R: Tableau(1..5);
begin
   if T'length<5 then
      raise constraint_error;
   else
      R:=T(T'last-4..T'last);
      return R;
   end if;
end cinq_derniers;
```

Un petit exemple pour illustrer la relation entre un tableau et une de ses tranches :

type TAB **is** array(Integer range <>) of Integer ;
T1 : TAB(1..3) := (**others** => 0) ;-- T1 initialisé avec que des 0
T2 : TAB := T1(2..3) ;-- initialisation de T2 avec la tranche de tableau T1(2..3), T2 est indexé par 2..3
...
T2(2) := 1 ;-- T2=(1,0), T1=(0,0,0), T1(2..3)=(0,0)
T1(3) := 4 ;-- T2=(1,0), T1=(0,0,4), T1(2..3)=(0,4)

On constate bien que T1(2..3) correspond bien aux cases 2 et 3 du tableau T1. Par contre, T2 est un tableau différent, lors de son initialisation les cases 2 et 3 de T1 sont recopiées dans les cases de T2, mais T1 et T2 ont des places disjointes en mémoire, donc les modifications de T1 n'affectent pas T2.

12.6 Opérations prédéfinies sur les tableaux

12.6.1 Comparaisons

On dispose pour les types tableaux des opérateurs de comparaison : =, /=, <, >, <=, >= à condition que le type des éléments du tableau en dispose aussi. L'ordre suivi est l'ordre lexicographique (même principe que l'ordre alphabétique) sur les cases du tableau :

- (2.0, 4.0) < (2.0, 6.0)
- (2.0, 4.0, 7.0) < (2.0, 6.0, 5.0)
- (2.0, 4.0) < (2.0, 4.0, 3.0)

En particulier si deux tableaux sont égaux, c'est qu'ils ont la même taille **et** que leurs éléments sont égaux deux à deux case par case.

12.6.2 Concaténation

L'opérateur `&` permet de concaténer deux tableaux de même type : `T1 & T2` donne un tableau de taille `T1'length+T2'length`, contenant les éléments de `T1` suivis de ceux de `T2`.

`&` permet également de concaténer un élément à un tableau :

```
T:Tableau(1..5);
T1:Tableau(1..2);
T2 : Tableau(1..4):=(others=>1);

T:=5&T2;  -- (5, 1, 1, 1, 1)
T:=5&4&T1&6; -- il faut au moins un tableau au milieu
T:=1&2&3&4&5; -- incorrect !
```

Attention ! Une fois déclaré, un tableau en ADA a une taille *fixe* : la mémoire est allouée une fois pour toute, et son nombre d'éléments ne peut ni augmenter, ni diminuer. Une instruction du type *T :=2&T* est donc totalement proscrite, puisqu'elle affecte un tableau de taille $n+1$ à un tableau de taille n.

12.7 Types tableaux anonymes

On peut définir un tableau simplement sans définir de type tableau :

T : array(1..5) of Float ;

Dans cas le type de T est anonyme (il n'a pas de nom) : c'est le type de T et d'aucun autre élément, on peut pas créer d'autre tableau du même type que T. Par exemple si on déclare :

T : array(1..5) of Float ;
S : array(1..5) of Float ;

S := T ;-- refusé par le compilateur, S et T ne sont pas de même type

Les paramètres de sous-programme ne peuvent pas être de type anonyme, la déclaration suivante est illicite :

function T : array(1..5) of Float(Float) **return** -- is interdit !

Il faut nécessairement définir un type pour pouvoir passer un tableau en paramètre d'un sous-programme.

12.8 Les chaînes de caractères

En ADA, une chaîne de caractères est simplement un tableau de caractères. Le type est défini comme un tableau **non contraint**. Il faut préciser l'intervalle lors de la déclaration d'une variable de type string. Plus précisément, le type `STRING` est un type pré-défini du paquetage `STANDARD` :

```
type STRING is array (POSITIVE range <>) of CHARACTER;
```

Les chaînes de caractères se manipulent donc exactement comme des tableaux. En particulier, une chaîne de caractères a une longueur fixe, qui ne varie plus après la déclaration de la chaîne :

```
S : STRING(1..5);
S : STRING := "je suis une chaîne de caractères !";
```

Quand on ne connait pas a priori la longueur d'une chaîne de caractères, on est obligé de fixer une borne supérieure lors de la déclaration, puis de gérer explicitement la longueur effective.

Et les entrées-sorties ? La procédure

```
Get_Line(Item: out STRING; Last: out NATURAL)
```

opère la saisie de la chaîne de caractères et affecte la longueur effective de celle-ci à la variable `Last`.

```
S : STRING(1..10);
L : NATURAL;
begin
   get_line(S,L);
     -- "Ernest" -> L vaut 6
   traitement(S(1..6));
end...
```

L'affichage se fait avec `put`.

12.9 Tableaux à plusieurs dimensions

Comme pour les tableaux à une dimension, il y a des tableaux multidimensionnels contraints et d'autres non contraints.

type Matrice43 **is** array(1..4,1..3) of Float ;-- 2 dimensions, contraint

type Matrice **is** array(Positive range <>, Positive range <>, Positive range <>) of Float ;-- 3 dimensions non contraint

M1 : Matrice43 ; ;-- pas d'intervalle à préciser, on les connaît déjà

M2 : Matrice(1..4,1..3) ;-- On précise l'intervalle pour chaque dimension, M1 et M2 sont de types différents ...

Attention, on ne peut pas mélanger contraint et non contraint : si une dimension est contrainte toutes les autres doivent l'être obligatoirement.

Pour l'accés aux éléments on doit simplement préciser la position respectivement à toutes les dimensions : M1(1,3) etc.

Il y a aussi des notations par agrégat :

M1 := ((2.0, 1.0, 3.0), (2=>4.0, others =>0.0), (others=>0.0), (3.0, others=>1.0)) ;

Il n'y a pas de tranche de tableau ni de concaténation pour les tableaux multidimensionnels.
Pour les attributs il faut préciser de quelle dimension on parle :
- Matrice'First(1) → 1
- Matrice'Last(1) → 4
- Matrice'Range(1) → 1..4
- Matrice'Length(1) → 4
- Matrice'First(2) → 1
- Matrice'Last(2) → 3
- Matrice'Range(2) → 1..3
- Matrice'Length(2) → 3

13 Les exceptions

Une *exception* est une situation inhabituelle, qui ne s'accorde pas avec le cas général prévu par le programme, la procédure ... et qui demande donc une gestion spécifique. L'avantage du mécanisme d'exception est qu'il permet un traitement uniforme des erreurs au sein du programme ADA.

13.1 Exceptions prédéfinies

`CONSTRAINT_ERROR` : activée quand on ne respecte pas une contrainte d'intervalle, de tableau ...

`STORAGE_ERROR` : problème de mémoire

`PROGRAM_ERROR` : activée quand on appelle un sous-programme inexistant, quand l'appel à une fonction se termine sans résultat retourné (`return`)...

`NUMERIC_ERROR` : activée quand une variable numérique est trop grande ...

La nature de l'exception déclenchée est une aide pour la correction de l'erreur de programmation. Il y également des exceptions prédéfinies liées aux entrées-sorties.

`ADA.IO_EXCEPTIONS.DATA_ERROR` : activée si la valeur lue avec un `get` ne correspond pas au format attendu.

```
N : Natural;
get(N);
      --   A -> raised ADA.IO_EXCEPTIONS.DATA_ERROR
      --  -2 -> raised CONSTRAINT_ERROR
```

Une exception, lorsqu'elle est déclenchée, se propage à travers tous les niveaux du programme, et en interromp l'exécution.

13.2 Déclaration d'exception

Il est possible de définir ses propres exceptions de la manière suivante :

```
<identificateur_exception> : exception;
```

La portée d'une exception est la même que pour toute autre variable ou constante, une exception n'existe que dans le bloc où elle est déclarée.

13.3 Levée d'exception

Une exception peut être déclenchée automatiquement au cours de l'exécution du programme mais peut aussi être déclenchée manuellement grâce à l'instruction :

```
raise <identificateur_exception>;
```

Comme lors d'un déclenchement automatique, l'exécution de l'instruction `raise` interromp le déroulement normale du programme et transfert le contrôle à un traitant de cette exception.

13.4 Traitement des exceptions

Le traitement d'une exception peut être prédéfini comme dans le cas des exceptions prédéfinies ou être codé par le programmeur. Il peut aussi être implicite, en l'abscence de traitant d'exception dans une procédure ou une fonction, le traitement d'une exception qui serait levée au cours de l'exécution de cette procédure ou fonction est simplement transféré à l'appelant.

Pour traiter une exception dans un bloc (délimité par un `begin` et un `end`), il suffit d'ajouter à la fin du bloc une partie `exception` :

```
begin
 <instructions>
exception
  when <identificateur_exception_1>=><traitement>;
  [when <identificateur_exception_2>=><traitement>;] -- en option
  [when others=><traitement>;]                       -- en option
end;
```

Si une exception est propagée en dehors du bloc où elle existe elle ne pourra être traitée que grâce au filtre `others` qui permet de traiter toutes les instructions qui n'ont pas encore été traitées.

Lorsque le déclenchement à lieux dans une partie déclarative, par exemple lors de l'initialisation d'une variable au moment de sa déclaration (appel de fonction qui léve une exception, expression comportant une division par zéro etc), l'exception n'est pas traité directement dans le bloc comportant la déclaration mais par celui qui l'appel.

Même chose lorsque l'exception est levée, automatiquement ou manuellement, dans un traitant d'exception : l'exception est propagée au bloc appelant.

14 Enregistrements

Définition Un enregistrement permet de définir des objets composés hétérogènes, contrairement aux tableaux qui sont homogènes : permet de regrouper en un seul objet des informations de type différents.

Le type d'un champ d'enregistrement est nécessairement un type contraint. Sinon, comment pourrait-on contraindre le type à la déclaration ?

Exemple

```
type JOUR is new Integer range 1..31 ;
type MOIS is (janvier, février, ... décembre) ;
type ANNEE is new Integer range 0..3000 ;
type DATE is record
        J : JOUR ;
        M : MOIS ;
        A : ANNEE ;
end record;
...
D : DATE ;
...
```

faire un petit dessin pour représenter un objet de type DATE

Accès aux champs

```
D1, D2 : DATE ;
D1.J := 22 ;
D1.M := octobre ;
D1.A := 2008 ;
D2.J := D1.J ;
D2.M := octobre ;
D2.A := D1.A ;
```

faire un petit dessin pour représenter D1 et D2

Comparaisons On dispose des opérateurs = et /= : compare champs par champs et retourne vrai (resp. faux) si les valeurs de chaque champs sont les mêmes pour les deux objets.

Agrégats - Affectations

```
D1, D2, D3 : DATE ;
D1 := (22,octobre,2008) ;
D2 := (A=>2008,J=>22,M=>octobre) ;
if D1=D2 then
        D3 := D1 ;
end if;
```

La première notation avec parenthèses n'est valable que pour les enregistrements comportant au moins deux champs. Pour les enregistrements à un seul champs, il faut utiliser la notation =>.

Valeur par défaut On peut donner une valeur par défaut aux champs à la déclaration du type :

```
type JOUR is new Integer range 1..31 ;
type MOIS is (janvier, février, ... décembre) ;
type ANNEE is new Integer range 0..3000 ;
type DATE is record
```

```
        J : JOUR := 22 ;
        M : MOIS := octobre ;
        A : ANNEE := 2008 ;
end record;
...
D : DATE ;
...
```

Exemple de fonction

```
function premier_jour_2008 return DATE is
begin
        return (J=>1,A=>2008,M=>1) ;
end premier_jour_2008;
```

15 Pointeurs et type access

15.1 Définition et déclaration

Définition : Un **pointeur** est un objet/une variable qui contient un accès/une référence/un lien vers un autre objet de type quelconque. C'est simplement une variable dont la valeur est égale à l'adresse d'une case mémoire.

En utilisant des pointeurs on peut définir des structures de données dynamiques, c'est-à-dire que leur taille et leur structure peut évoluer au cours du temps et n'être pas connue à la compilation, comme des listes, des arbres etc.

Il y a deux sortes de pointeurs en Ada : accès à des objets et accès à des sous programmes (permet de faire de l'ordre supérieur).

Déclarations : Soit `T` un nom de type ou de sous type, on peut déclarer un type pointeur vers les objets de type `T` :

```
type ptrT is access T ; -- un objet de type ptrT a pour valeur un accès à un objet de type T
X,Y : ptrT ;
```

Pour l'instant `X` et `Y` ne pointent sur rien : par défaut, une fois déclaré, un pointeur est initialisé à la constante `null`, qui ne correspond à aucune adresse. La valeur `null` est commune à tous les types accès.

```
X := null ;
return Y = null ; -- retourne TRUE , Y n'étant pas initialisé il vaut null
```

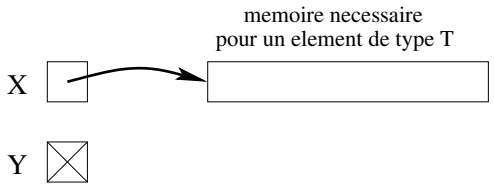
15.2 Allocation dynamique de mémoire.

La déclaration `X,Y : ptrT` a pour effet d'attribuer de la mémoire aux variables `X` et `Y`, comme n'importe quelle déclaration de variable. Mais, il n'y a pas d'allocation de mémoire pour l'objet pointé, ici de type `T`. Il faut une allocation explicite, avec l'instruction `new` qui prend en argument le type de l'objet.

```
type ptrT is access T ;
X,Y : ptrT ;
X := new T ;
```

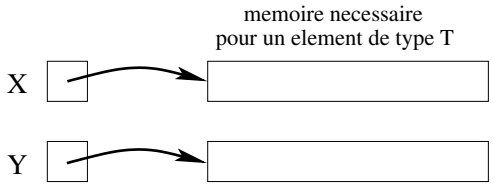
L'expression `new T` a deux effets :

1. réserve une zone mémoire pour un objet de type `T`
2. retourne pour valeur un accès à cette zone mémoire, c'est-à-dire à cet objet

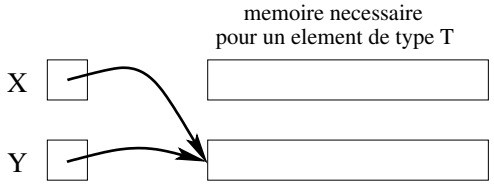


Exercice 1 Affectation de pointeurs
Représenter par un schéma ce qui se passe.

Q1 . Y := **new** T



Q2 . X := Y



`X` et `Y` pointent maintenant vers le même objet de type `T`, l'objet pointé par X avant cette affectation est perdu, on n'a plus d'accès vers lui.

15.3 Accès à l'objet pointé : déréférencement

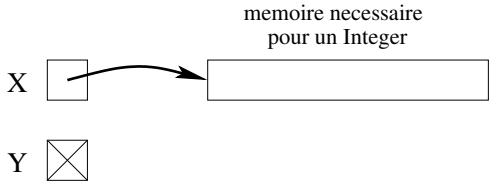
Si `X` est un accès à un objet de type `T` alors `X.all` est cet objet de type `T`. Si `X = null` on aura une erreur lors de l'évaluation de `X.all` à l'execution.

Exercice 2 Accès à l'objet pointé

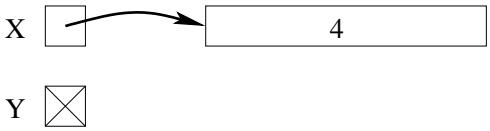
Représenter par un schéma ce qui se passe. On définit :

type ptrInt **is access** Integer ;
X,Y : ptrInt ;

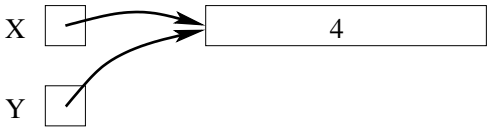
Q1 . X := **new** Integer



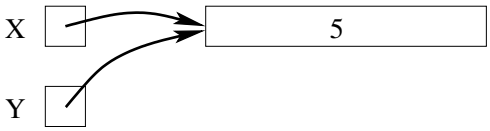
Q2 . X.all := 4



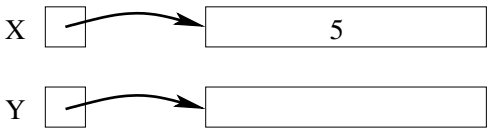
Q3 . Y := X



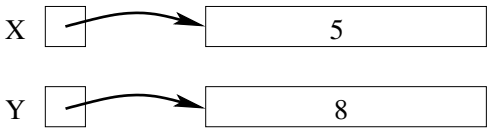
Q4 . Y.all := Y.all+1



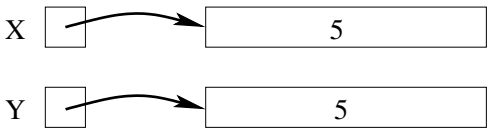
Q5 . Y := **new** Integer



Q6 . Y.all := 8



Q7 . Y.all := X.all



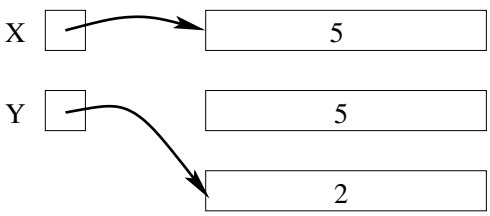
Q8 . Qu'est ce qui est comparé lorsqu'on fait `X = Y` et `X.all = Y.all` ?

15.4 Allocation initialisée

Il est possible de combiner l'allocation de mémoire et l'initialisation de l'objet abrité dans cette case mémoire en une seule instruction, avec la syntaxe suivante :

```
X : ptrT ;
Y : ptrInt ;
...
X := new T'(<expression de type T>) ;
Y := new Integer'(4+3) ; -- Y.all = 7
```

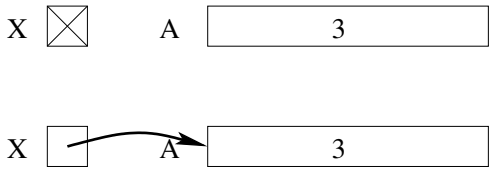
Exercice 3 Réaffectation
En partant de la situation obtenue à la fin de l'exercice précédent, représenter par un schéma ce que donne l'instruction `Y := new Integer'(2)`



15.5 Pointeur sur une variable déjà déclarée

On peut récupérer un accès à une variable avec l'attribut Access :

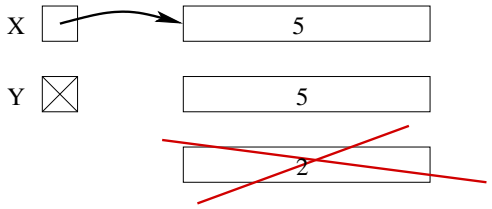
```
type ptrInt is access Integer ;
X : ptrInt ;
A : Integer := 3 ;
...
X := A'Access ;
```



15.6 Désallocation/libération de mémoire

Qui dit *allocation* dit ensuite *désallocation*, ou *libération* de mémoire. Cela se fait par instanciation de la procédure générique `Unchecked_Deallocation`.

```
with Unchecked_Deallocation ;
...
procedure Free is new Unchecked_Deallocation(Integer, ptrInt) ;
...
Free(Y) ; -- la valeur pointée par Y est perdue, la mémoire est libérée
```



15.7 Conversion de types accès

```
type ptrTX is access T ;
type ptrTY is access T ;
X : ptrTX ;
Y : ptrTY ;
X := Y ; -- impossible, pas le même type
X := ptrTX(Y) ; -- impossible aussi
```

Pour pouvoir convertir `Y` en pointeur de type `ptrTX` il faut que le type `ptrTX` soit déclaré comme un type accès à objet général :

```
type ptrTX is access all T ;
type ptrTY is access T ;
X : ptrTX ;
Y : ptrTY ;
X := ptrTX(Y) ; -- maintenant ça marche
```

15.8 Passage de paramètre de type accès

Si on déclare une procédure `procedure P(X : access T) is ...` X est un paramètre de type "pointeur sur T" anonyme. Le paramètre effectif doit être non null au moment de l'appel de la procédure et peut être de n'importe quel type accès à T. Il faut convertir en un autre type pour pouvoir faire une affectation à un autre pointeur d'un type nommé.