Aide-mémoire Ada (2)

Algorithmique, second semestre

Domaine mathématiques et algorithmique

INSA première année

Structures de données

NOM :

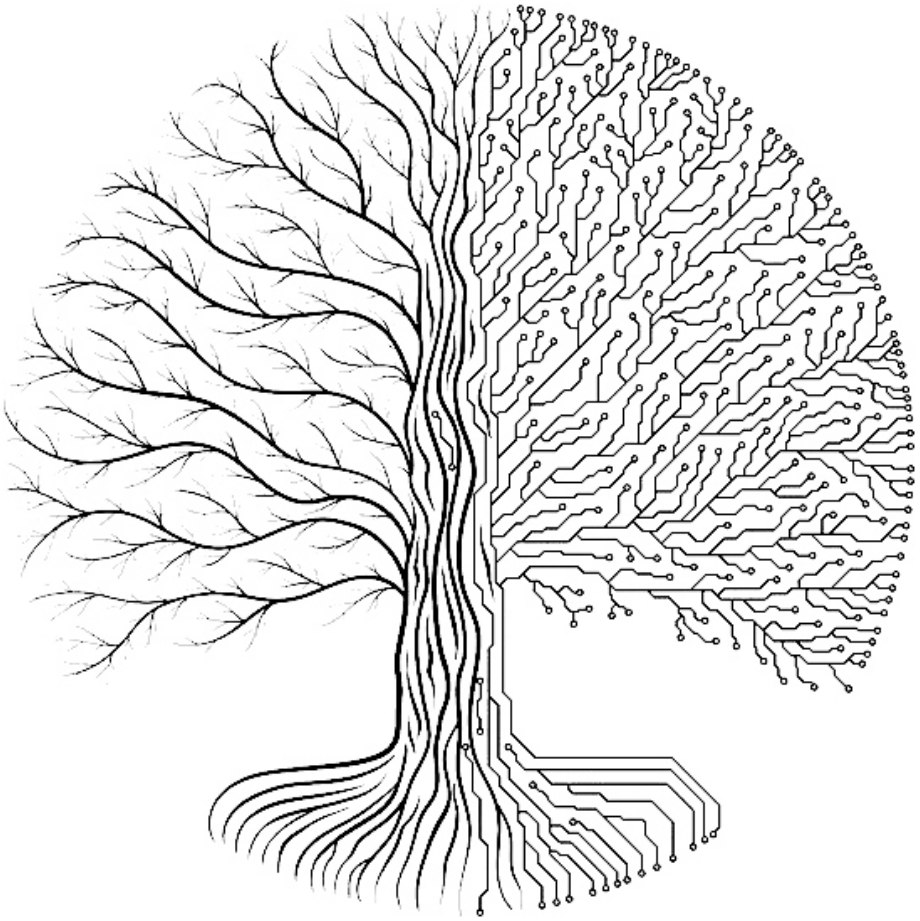
.....

PRÉNOM :

.....

GROUPE :

.....



D. LE BOTLAN

contact.lebotlan@insa-toulouse.fr

Introduction

Alors que l'algo du premier semestre couvre la base de la programmation, il manque les outils de *passage à l'échelle* : comment écrire un programme qui gère un génome de 3 milliards de paire de bases, ou un réseau routier comprenant des millions de segments ?

Nous nous focalisons désormais sur la représentation des informations et sur la décomposition d'un problème complexe en tâches simples.

Notations

- La notation $e \in \tau$, où τ est un type, signifie que l'expression e a le type τ.
- De même $B \in \mathfrak{bloc}$ signifie que B est un bloc de code
- Enfin, $D \in \mathfrak{definition}$ signifie que D est une définition, et doit donc être placée avant le **begin**.

Table des matières

Règles de qualité

Afin d'améliorer la robustesse des programmes, il convient de respecter certaines normes de qualité (issues pour la plupart de la norme CNES pour Ada).

Règles de nommage

- Un identificateur est aussi informatif que possible, sans être trop long.
- Un nom de type commence par T_.
- Un nom de procédure commence en général par un verbe.

Règles esthétiques

- Le code est **indenté** (décalage judicieux des blocs, automatique dans emacs).
- Le code est aéré, mais sans excès.
- Des commentaires **perspicaces** aident à la lecture du code.

Règles de conception

- Définir des constantes si besoin : à part dans les tests, il ne doit pas y avoir de nombres après le **begin** (sauf éventuellement 0 ou 1).
- Utiliser une boucle **for** chaque fois que c'est possible, plutôt qu'une boucle **while**.
- Le **return** d'une fonction se situe à la fin de la fonction.
- Chaque fonction ou procédure majeure doit être testée avec une procédure associée : la fonction Foo est testée avec la procédure Tester_Foo.

Exemples d'identificateurs

Nombre_Mots	**procedure** Trouver_Min	**type** T_Gnome
Largeur	**procedure** Afficher_Ensemble	**type** T_No_Telephone
~~N~~	~~**procedure** Message~~	~~**type** Gnome~~
~~ZKLFS~~	~~**procedure** Trop_Grand~~	~~**type** Justin_Bieber~~

- X et Y sont des noms acceptables pour des coordonnées.
- Lorsque le rôle d'une procédure est clair, son nom n'est pas forcément un verbe. Par exemple Pause(10).

Sous-types

Un **sous-type** représente un **intervalle** dans un type existant.

Définition de sous-types

subtype T_Foo **is** *un_type_existant* **range** *intervalle* ;

$\in$ definition

Le sous-type peut ensuite être utilisé comme un type normal. Par exemple, pour déclarer une variable : Compteur : **T_Foo** ;

Exemples de sous-types

DÉFINITION DU SOUS-TYPE ($\in$ definition)	INTERVALLE REPRÉSENTÉ
subtype T_Foo **is** Integer **range** -50 .. 2000 ;	$[-50\ ;2000]$
subtype T_Bar **is** Float **range** 0.0 .. 1.0 ;	$[0.0\ ;1.0]$
subtype T_Moo **is** Character **range** 'A' .. 'D' ;	'A', 'B', 'C', 'D'
subtype T_Bla **is** Integer **range** Integer'First .. -40 ;	$]-\infty\ ;-40]$
subtype T_Foo **is** Integer **range** 0 .. Integer'Last ;	$[0\ ;+\infty[$

Noter qu'en Ada ∞ n'existe pas. Le plus grand entier vaut environ 2.10^9
Même remarque pour les réels (*Float*) : le plus grand réel vaut 3.4^{38}

Les sous-types **Natural** ($[0\ ;+\infty[$) et **Positive** ($[1\ ;+\infty[$) sont prédéfinis en Ada.

Notes personnelles

Types énumérés

Un **type énuméré** représente un ensemble fini de valeurs.

Définition d'un type énuméré

```
type T_Jour_Semaine is (Lun, Mar, Mer, Jeu, Ven, Sam, Dim) ;
```

$\in$ definition

Un type énuméré s'utilise comme tout type ou sous-type :

```
Demain : T_Jour_Semaine ;
```

Le compilateur détecte automatiquement les incohérences :

```
Demain := Mer ;
```
Correct

~~`Demain := Mercredi ;`~~ *Erreur de compilation (*Mercredi *est inconnu)*

RÈGLE *« Type énuméré »*

Après une définition **type** T_Foo **is** (Nom_1, Nom_2, …, Nom_k) ;
les k jugements $\text{Nom}_i \in$ *T_Foo* sont valides pour $1 \leqslant i \leqslant k$.

☞ Ainsi, on a Dim $\in$ *T_Jour_Semaine* (à comparer avec "Dim" $\in$ *String*).

Pour tester une variable d'un type énuméré, on utilise très souvent un bloc **case**.

Notes personnelles

Le bloc **case** permet de tester **un type énuméré**.

Définition du bloc **CASE**

Syntaxe :

```
case e is
   when Nom_1 => B_1 ;
      ⋮
   when Nom_k => B_k ;
end case ;
```

$B_i \in$ bloc

$e \in \tau$, où τ est un type énuméré.

$\text{Nom}_k \in \tau$

Exécution du bloc **CASE** :

1 – L' expression e est calculée et donne une valeur Nom_j .
2 – Le bloc B_j correspondant est exécuté, puis le bloc **case** se termine.

☞ Il est possible de factoriser plusieurs cas en écrivant **when** Nom_1 | Nom_2

RÈGLE « *Bloc* **CASE** »

Un bloc **CASE** est un bloc :

Si $e \in \tau$ $\text{Nom}_i \in \tau$ et $B_i \in$ bloc,

alors

```
case e is
   when Nom_i => B_i
   ...
end case ;
```

$\in$ bloc

Exemple d'utilisation

```
1  -- Cette procédure affiche son argument de type T_Jour_Semaine (voir page de gauche).
2  -- On suppose que l'acteur Txt est défini.
4  procedure Afficher_Jour (Jour : T_Jour_Semaine) is
5  begin
6     case Jour is
7        when Lun              => Txt.Put ("Lundi, jour difficile") ;
8        when Mar | Mer | Jeu  => Txt.Put ("Milieu de la semaine") ;
9        when Ven              => Txt.Put ("Vendredi") ;
10       when Sam | Dim        => Txt.Put ("Week-end") ;
11    end case ;
12 end Afficher_Jour ;
```

$\in$ bloc

$\in$ definition

☞ L' expression e testée par le bloc **case** est l'argument Jour (ligne 6).

Jour : *T_Jour_Semaine* ⟶ Afficher_Jour

Tableaux

Un **tableau** (*array*) contient un nombre fini de cellules du même type repérées par un indice : $v(1)$, $v(2)$, …, $v(k)$.

Définition d'un type tableau

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

$\in$ definition

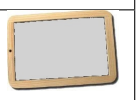
Ce type T_Foo permet de définir des tableaux de réels :

```
-- Bar est un tableau de 5 cellules réelles, numérotées de 1 à 5
Bar : T_Foo (1..5) ;
-- Moo est un tableau de 3 cellules réelles, numérotées de 10 à 12
-- et initialisées à 0.0
Moo : T_Foo (10..12) := (others => 0.0) ;
```

$\in$ definition

Accès aux cellules du tableau

L'expression Bar(*i*) , où *i* est un entier, est la i^{eme} cellule du tableau.
Pour modifier sa valeur, on écrit : Bar(*i*) := *valeur* ;

Bar	1	2	3	4	5
					

Moo	10	11	12
			

RÈGLE « *Tableau* »

Après la définition **type** T_Foo **is array** (*Integer* **range** <>) **of** τ où τ est un type quelconque,

- Si $e_1, e_2 \in$ *Integer* alors Bar : T_Foo (e_1 .. e_2) $\in$ definition
- Si $e \in$ *T_Foo* et $e' \in$ *Integer* alors $e(e') \in \tau$
- Si $e \in$ *T_Foo* et $e' \in$ *Integer* et $e'' \in \tau$ alors $e(e')$:= e'' $\in$ bloc

Parcours de tableaux

Pour parcourir toutes les cellules du tableau, on utilise une boucle **for**.
Les indices de début et de fin du tableau sont obtenus avec les attributs 'FIRST et 'LAST.

Parcours d'un tableau

```
for Index in Bar'First..Bar'Last loop
   -- Ici on ajoute 1.0 à toutes les cellules
   Bar(Index) := Bar(Index) + 1.0 ;
end loop ;
```

L'intervalle peut s'écrire de manière équivalente `Bar'Range`

Le nombre de cellules du tableau est obtenu avec `Bar'Length` .

RÈGLE *« Attributs d'un tableau »*

Après la définition **type** T_Foo **is array** (*Integer* **range** <>) **of** τ où τ est un type quelconque,

Si $e \in$ *T_Foo* alors e'First $\in$ *Integer* e'Last $\in$ *Integer* et e'Length $\in$ *Integer*

Notes personnelles

Matrices – Tableaux à deux dimensions

En Ada, une matrice de n lignes et m colonnes se représente par un **tableau** (*array*) à deux dimensions.

Définition d'un tableau à 2 dimensions

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

∈ definition

```
-- Bar est une matrice de 2 lignes et 4 colonnes
Bar : T_Matrice(1..2, 1..4) ;
-- Moo est une matrice de 3 lignes et 3 colonnes initialisée à 0
Moo : T_Matrice(5..7, 0..2) := (others => (others => 0.0)) ;
```

∈ definition

Accès aux cellules de la matrice

L'expression `Bar(i, j)`, où i et j sont des entiers, renvoie la cellule située ligne i, colonne j.
Pour modifier sa valeur, il suffit d'écrire : `Bar(i, j) := valeur ;`

Bar :

	1	2	3	4
1				
2				

Moo :

	0	1	2
5			
6			
7			

RÈGLE « *Matrice* »

Après la définition `type T_Mat is array (Integer range <>, Integer range <>) of` τ
où τ est un type quelconque,

- Si $e_1, e_2, e_3, e_4 \in$ *Integer* alors `Bar : T_Mat (`e_1 `..` e_2`,` e_3 `..` e_4`)` ∈ definition
- Si $e \in$ *T_Mat* et $e_1, e_2 \in$ *Integer* alors $e(e_1, e_2) \in \tau$
- Si $e \in$ *T_Mat* , $e_1, e_2 \in$ *Integer* et $e' \in \tau$ alors $e(e_1, e_2) := e'$ ∈ bloc

Parcours de matrices

Pour parcourir toute la matrice ligne par ligne, il suffit d'imbriquer deux boucles **for** (comparer avec le parcours de tableaux, page précédente).

Les indices des lignes ou des colonnes sont obtenus avec les attributs 'RANGE(1) et 'RANGE(2), respectivement.

Parcours d'une matrice

```
for Ligne in Bar'First(1)..Bar'Last(1) loop
   for Colonne in Bar'First(2)..Bar'Last(2) loop
      -- On met toutes les cellules à 0
      Bar(Ligne, Colonne) := 0.0 ;
   end loop ;
end loop ;
```

Les deux intervalles peuvent s'écrire de manière équivalente Bar'Range(1) et Bar'Range(2).

Bar'Length(1) donne le nombre de lignes de la matrice.

Bar'Length(2) donne le nombre de colonnes de la matrice.

RÈGLE *« Attributs d'une matrice »*

Après la définition **type** T_Mat **is array** (*Integer* **range** <>, *Integer* **range** <>) **of** τ

Si $e \in$ *T_Mat* alors e'First(1) $\in$ *Integer* et e'Last(1) $\in$ *Integer*

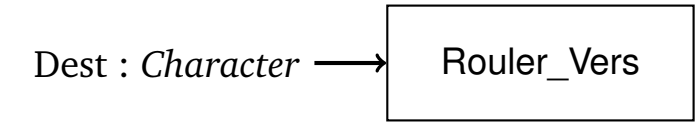
De même pour e'First(2) e'Last(2) e'Length(1) et e'Length(2)

Notes personnelles

Mode de passage IN OUT

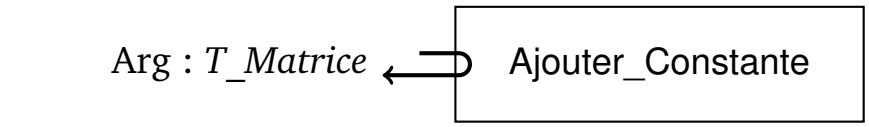
Par défaut, il est interdit de modifier les arguments des procédures et fonctions. On dit que les arguments sont en mode **IN**, c.-à-d. en entrée seulement.

Mode **IN**



Il est possible de définir des procédures qui modifient leur(s) argument(s), en particulier lorsque ce sont des tableaux ou des matrices.
Un argument modifié est en mode **IN OUT**.

Mode **IN OUT**



Lors de l'exécution de la procédure, la matrice passée en argument est modifiée.

Notes personnelles

Procédures avec arguments IN OUT

Une procédure peut modifier ses arguments qui sont en mode **IN OUT**.
(en général des tableaux ou des matrices)

Définition de procédure avec argument **IN OUT**

```
procedure Foo (Bar1 : in Float ; Bar2 : in out T_Matrice) is
begin
   -- Corps de la procédure
end Foo ;
```

Ce **in** est facultatif

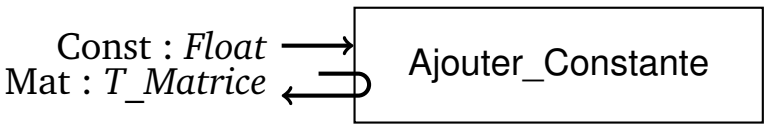
∈ definition

Exemple de procédure avec argument **IN OUT**

```
-- Cette procédure ajoute une constante à chaque cellule de la matrice
procedure Ajouter_Constante (Const : in Float ; Mat : in out T_Matrice) is
begin
   for Ligne in Mat'Range(1) loop
      for Col in Mat'Range(2) loop
         Mat(Ligne, Col) := Mat(Ligne, Col) + Const ;
      end loop ;
   end loop ;
end Ajouter_Constante ;
```

L'argument Mat est en mode **IN OUT**. Il est **modifiable**

∈ definition



☞ **UNE PROCÉDURE AVEC ARGUMENTS IN OUT N'EST PAS UNE FONCTION !**

(Elle ne renvoie rien)

Un bloc **declare** permet d'insérer des définitions dans un bloc quelconque.

RÈGLE « *Bloc* **DECLARE** »

Si $D_j \in$ definition, et $\mathcal{B} \in$ bloc,

alors
declare
D_j ;
begin
$\mathcal{B}$;
end ;
$\in$ bloc

Les définitions D_j sont évaluées

puis le bloc $\mathcal{B}$ est exécuté.

Exemple de bloc **declare**

```
with GAda.Text_IO, GAda.Integer_Text_IO ;
procedure Mission is
   package Txt renames GAda.Text_IO ;
   package ITxt renames GAda.Integer_Text_IO ;
   type T_Entiers is array(Integer range <>) of Integer ;
   Nb_Termes : Integer ;
begin
   Txt.Put("Combien de termes a entrer ? ") ;
   Nb_Termes := ITxt.FGet ;
   declare
      ZeTab : T_Entiers (1..Nb_Termes) ;          ∈ definition
   begin
      for Indice in ZeTab'Range loop
         Txt.Put("Entrer le terme numero " & Indice'Image & " : ") ;
         ZeTab(Indice) := ITxt.FGet ;
      end loop ;
   end ;                                                  ∈ bloc
end Mission ;
```

SOUS-TYPE (p. 3)

subtype T_Foo **is** *un_type_existant* **range** *intervalle*

ex : **subtype** Natural **is** Integer **range** 0 .. Integer'**Last**

TYPE ÉNUMÉRÉ (p. 4)

```
type T_Signal is (Rouge, Orange, Vert) ;
-- Déclaration d'une variable
Foobar : T_Signal ;          ∈ definition
-- Affectation de la variable
Foobar := Orange ;           ∈ bloc
```

BLOC **CASE** (p. 5)

```
case Foobar is
   when Rouge  => B1 ;
   when Orange => B2 ;
   when Vert   => B3 ;
end case ;                   ∈ bloc
```

TABLEAUX (p. 6)

```
-- Définition d'un type tableau
type T_Foo is array (Integer range <>) of Float;
-- Déclaration d'un tableau de 5 cellules
Bar : T_Foo (1..5) ;
-- Affectation des cellules du tableau
Bar(1) := 12.0 ;
Bar(2) := Bar(1) * 3.0 ;
```

PARCOURS DE TABLEAUX (p. 7)

```
for Num in Bar'Range loop
   Bar(Num) := Bar(Num) + 1.0 ;
end loop ;
```

MATRICES (p. 8)

```
-- Définition d'un type matrice
type T_Mat is array (Integer range <>,
                     Integer range <>) of Float;
-- Déclaration d'une matrice 2x3
Moo : T_Mat (1..2, 1..3) ;
-- Affectation des cellules de la matrice
Moo(1,1) := 0.0 ;    -- Ligne 1, colonne 1
Moo(2,1) := 100.0 ;  -- Ligne 2, colonne 1
```

PARCOURS DE MATRICE (p. 9)

```
for Ligne in Moo'Range(1) loop
   for Col in Moo'Range(2) loop
      Moo(Ligne, Col) := 5.0 ;
   end loop ;
end loop ;
```

BLOC **DECLARE** (p. 12)

```
declare
   Dj ;  ∈ definition
begin
   B ;   ∈ bloc
end ;
```

PROCÉDURE AVEC ARGUMENT **IN OUT** (p. 11)

```
procedure Foo (Bar1 : in Float ; Bar2 : in out T_Mat) is
begin
   -- Corps de la procédure
end Foo ;
```

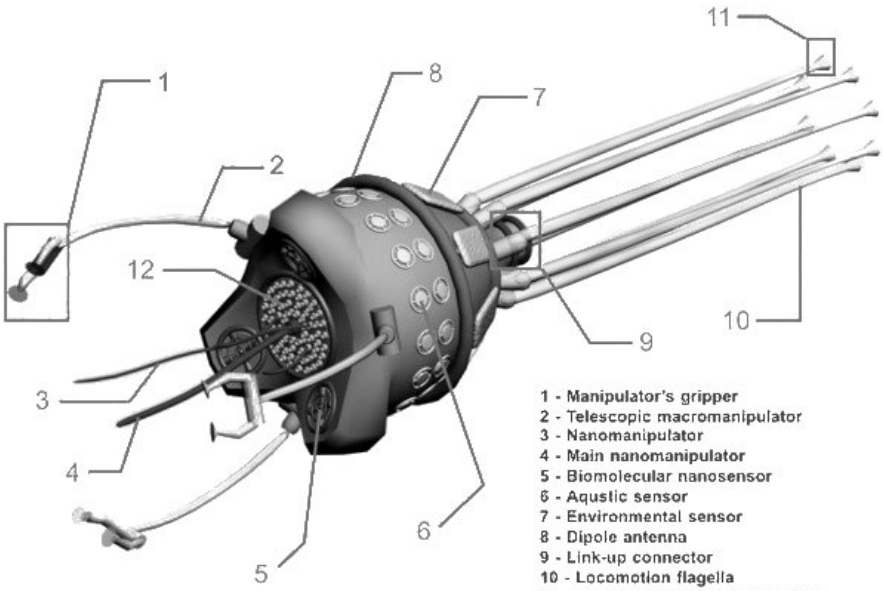


Image courtesy of Nanotechnology News Network
Author: analyst Svidinenko Yuriy

Document compilé le 4 décembre 2024.