Algorithmique & Génie Logiciel avec ADA

Une Introduction

Version 1.0 , le jeudi 23 août 2007

La version la plus récente est disponible à http://www.dgaudry.com

Pour toute correspondance, utiliser : daniel@dgaudry.com

Download :

HTTP download open office version: VERSION OPEN OFFICE

HTTP download pdf version: Version PDF

Table des matières

1 INTRODUCTION AU LANGAGE

Ce langage a été développé pour permettre la traduction directe de l'analyse intellectuelle d'un problème dans un code sans étape intermédiaire. Il permet de coder depuis l'application la plus simple jusqu'au code orienté objet multi-tâche. Il permet la capture complète au niveau de chaque élément du code des contraintes issues de l'analyse du problème. Ada permet également un traitement efficace et élégant des erreurs survenues pendant le déroulement du programme.

2 LA STRUCTURE DE BASE

La structure de base d'un langage est la grammaire (avec sa ponctuation) et le vocabulaire. La grammaire gouverne l'ordre des mots dans la 'phrase' et le vocabulaire est défini par:

2.1 CE QU'IL FAUT SAVOIR

2.1.1 Les mots réservés

- Des mots, déjà créés et connus du compilateur, dits **réservés:**
  Leur signification est déjà fixée par le langage
  Il ne peuvent pas servir de nom de variable ou de fonction / procédure :

abort	else	new	return
abs	elsif	not	reverse
abstract	end	null	
accept	entry		select
access	exception		separate
aliased	exit	of	subtype
all		or	synchronized
and	for	others	tagged
array	function	out	task
at		overriding	terminate
	generic	package	then
begin	goto	pragma	type
body		private	
	if	procedure	
case	in	protected	until
constant	interface	raise	use
declare	is	range	when
delay	limited	record	while
delta	loop	rem	with
digits		renames	
do	mod	requeue	xor

2.1.2 Les attributs

- Des 'Attributs' qui définissent les propriétés des variables:
  liste des principaux attributs :

access	address	adjacent	aft
alignment	all	base	bit_order
body_version	callable	caller	ceiling
class	component_size	compose	constrained
copy_sign	count	definite	delta
denorm	digits	exponent	external_tag
first	first_bit	floor	fore

fraction	identity	image	input
last	last_bit	leading_part	length
machine	machine_emax	machine_mantissa	machine_overflows
machine_radix	machine_rounds	max	max_size_in_storage_elements
min	model_emin	model_epsilon	model_mantissa
model_small	modulus	new	output
pos	pred	range	read
remainder	round	rounding_safe_first	safe_last
scale	scaling	signed_zeros	size
small	storage_size	succ	tag
terminated	truncation	unbiased_rounding	unchecked_access
val	valid	value	version
wide_image	wide_value	wide_width	width
write			

2.1.3 Des noms créés par l'utilisateur

Il suivent une règle bien précise (voir plus loin), et servent à identifier les variables, le noms de procédure et les noms de fonctions.
Quelques exemples (en ada il n'y a pas de différence entre majuscule et minuscule):
A
bonjour
Ma_feuille
a_1

2.1.4 Des lignes de code

Chaque 'phrase' se termine par ;

```
a := 1;
dsort(a => b);
x := 1 + X;
```

2.1.5 Des structures

Ces structures (procédure, boucle, test if, ….) récursives qui s'emboîtent un peu à la façon des poupées gigognes, elles se terminent SYSTÉMATIQUEMENT par **END (loop, if, ….);**

2.1.6 Des opérateurs

Assignation	:=
Addition	+
Soustraction	-
multiplication	*
Division	/
Puissance	**
Et	and
Ou	or
Non	not.
Ou exclusif	xor
Supérieur	>
Inférieur	<
Supérieur ou égal.	>=
Inférieur ou égal.	<=
égal	=
Différent	/=
Concaténation (de tableaux, de chaîne de caractères)	&
Choix multiple. (dans les «case» et les initialisation de tableaux)	|
Explicitation de paramètres dans un appel fonction/procédure	=>

2.2 CRÉATION D'UN PROGRAMME

Pour créer un programme il faut:

2.2.1 Le fichier texte contenant le code source

Créer un fichier texte résidant sur le disque de votre ordinateur, par exemple aaa.adb (noter l'extension adb, et le nom aaa) dont voici un exemple valable (qui ne fait rien !). Il est divisé en plusieurs zones:

```
--zone 1
With bbb;


Procedure aaa
Is

--zone 2

Begin

--zone 3
Null;


End aaa;
```

Les lignes de code commençant par – sont des commentaires non pris en compte.

2.2.2 Zone 1

C'est ici que l'utilisateur déclare les procédures, fonctions et packages (qui sont des collections de procédures et de fonctions) contenues dans D'AUTRES FICHIERS ou déjà incorporés au langage. Cette zone s'arrête au mot clef «**procedure»** (ou «**package»** ou «**function»**).

2.2.3 Zone *2*

Dans cette zone on déclare:

- Les types de variables.
- Les variables à partir de leur type.
- Des procédures et/ou fonctions à l'usage exclusif de la procédure aaa.
- Les limites d'utilisation des variables et des types de variables déclarés dans des packages extérieurs en ce qui concerne la comparaison.
- Les zones de validité des variables dans le cas d'usage de procédures et / ou fonctions à l'usage exclusif de la procédure aaa.

2.2.4 Zone 3

C'est là que se trouve le code source qui **D O I T** être segmenté en parties indépendantes. Ce code doit être ***ABONDAMMENT COMMENTÉ*** pour rendre sa logique lisible et facile à suivre.

2.3 LE CODE SOURCE ET SA TRADUCTION EN EXÉCUTABLE

Une fois le code écrit il faut le traduire dans un programme par un procédé qui s'effectue par:

2.3.1 La compilation

Le compilateur vérifie d'abord que les règles de grammaire et de syntaxe sont respectées et traduit le code source en un fichier 'objet' qui représente, pour simplifier, le code source en un code machine intermédiaire.

2.3.2 Le link

Le linker transforme le fichier objet et tous les fichiers objet associés dans la zone 1 (et ceux implicitement associes) en un programme que l'on peut exécuter. Il faut, dés le début, prendre comme habitude de créer des structures complètes et de demander **SYSTÉMATIQUEMENT AU COMPILATEUR DE TESTER LA SYNTAXE**.

2.3.3 L'éditeur

L'utilisation d'un éditeur spécialisé, et de conception moderne, permet en outre une productivité accrue, et facilite beaucoup l'apprentissage.

Les possibilités de l'éditeur **doivent** comprendre:

- Copier/coller par raccourci clavier.
- Macro pour automatiser les tâches d'édition répétitives.
- Codage coloré pour les différentes partie de la syntaxe.
- Sauvegarde automatique de l'état d'avancement du travail.
- Retour au même endroit après fermeture et ouverture de l'éditeur.
- Mise en forme automatique du code pour faciliter la lecture.
- Une grammaire élémentaire pour faciliter la recherche des erreurs de syntaxe.

2.3.4 Un premier exemple

Tous les détails seront expliqués plus tard, voici le texte du code stocké dans le fichier premier.adb et le résultat après compilation, link et exécution:

```
with Ada.Text_Io;

procedure Premier
  is
  Resultat : Integer;

  begin
  Resultat := 2;
  Resultat := Resultat ** 10;
  Ada.Text_Io.Put_Line("2 puissance 10 = " & Integer'Image(Resultat));

end Premier;
```

La compilation se fait directement depuis l'éditeur ou en ouvrant un fenêtre «dos» ou «xterm» selon l'O.S. employé, puis en utilisant la commande
gnatmake premier

Utilisation du compilateur GNAT (fenêtre DOS sous windows)

```
F:\ada\current\w\miage>gnatmake premier
gcc -c premier.adb
gnatbind -x premier.ali
gnatlink premier.ali
F:\ada\current\w\miage>
```

et le résultat:

```
F:\ada\current\w\miage>premier
2 puissance 10 =  1024
F:\ada\current\w\miage>
```

Utilisation du compilateur GNAT (fenêtre xterm sous linux)

```
[daniel@localhost miscellaneous]$ gnatmake premier
gcc -c premier.adb
gnatbind -x premier.ali
gnatlink premier.ali
[daniel@localhost miscellaneous]$ ./premier
2 puissance 10 =  1024
[daniel@localhost miscellaneous]$
```

D'autres compilateurs peuvent aussi être utilisés (Janus, Aonis,……)

Quelques détails pour les curieux:

Sauf entre " et " les espaces et les retours à la ligne sont ignorés et sont seulement utilisés pour **rendre le code lisible.**

3 LES VARIABLES

Tous les langages de programmation modernes imposent une déclaration préalable des variables (basic n'est pas, de ce point de vue, un langage moderne).
ADA impose les règles suivantes pour la création de nom (variable, type de variable, nom de fonction de procédures de package, …….)

Les règles pour inventer un nom valable (identifier) sont décrites comme suit:

Identifier	**::=**	identifier_letter **{ [** underline **]** letter_or_digit **}**
letter_or_digit	**::=**	identifier_letter **|** digit
identifier_letter	**::=**	upper_case_identifier_letter **|** lower_case_identifier_letter
digit	**::=**	'0' **|** '1' **|** '2' **|** '3' **|** '4' **|** '5' **|** '6' **|** '7' **|** '8' **|** '9'
underline	::=	'_'

Les abréviations (généralement utilisées en algorithmique pour décrire une grammaire) sont les suivantes:

{ xxxxxx **}**	veut dire	0 à n répétitions de xxxxxx
[yyyyyyy **]**	veut dire	yyyyyy est optionnel
|	veut dire	**ou**
::=	veut dire	défini par
upper_case_identifier_letter	veut dire	toutes les lettres majuscules de A à Z
lower_case_identifier_letter	veut dire	toutes les lettres Minuscules de a à z

Cette définition servira, dans un exercice, à illustrer la capacité du langage à exprimer directement un problème posé (illustration simple d'un automate à états finis contre des fonctions récursives avec test if imbriqués et boucles) .

3.1 LES DIFFÉRENTS TYPES DE VARIABLES

Ada permet de représenter tous les objets que l'on peut rencontrer dans l'écriture d'un programme.
Certains types de variables sont déjà prédéfinis: les nombres, les tableaux et matrices, les caractères, les chaînes de caractères.
D'autres sont crées par l'utilisateur selon ses besoins.
Nous allons voir systématiquement comment effectuer ces déclarations.
Chaque type a des attributs qui lui sont associés et décrivent ses propriétés. C'est un point important du langage car ces 'attributs' participent à l'écriture du code en facilitant sa compréhension et sa sûreté de fonctionnement

3.2 LES DÉCLARATIONS DE TYPE DE VARIABLES ET LES DÉCLARATIONS DE VARIABLES À PARTIR DE LEUR TYPE

3.2.1 Les nombres constants

Dans ce cas on sait que la valeur ne changera pas (d'ailleurs le compilateur y veillera)

```
Charge_de_rupture : constant := 8.0 * 1024.0;
Max               : constant := 500;
Taille_de_tableau : constant := Max / 6
memory            : constant := 2 ** 16;
a, b ,c           : constant := 1;
```

Le mot clé **constant** sera utilisé dans toutes les déclarations si le contenu de l'objet ne doit pas être modifié dans le programme.

3.2.2 Les entiers

Un type entier est prédéfini et s'appelle integer (ou long_integer)

3.2.2.1 Integer ou long_integer

Ce sont deux types prédéfinis avec les limites suivantes (**peuvent varier d'un compilateur à l'autre et aussi en fonction du CPU**)
Integer varie de -2^{32} à 2^{32} -1 (par exemple)
Long_integer varie de -2^{64} à 2^{64} -1 (par exemple);
L'instruction:

```
Z: integer;
```

Défini une variable dont le nom est Z pour stocker des valeurs de -2^{32} à 2^{32} -1 (par exemple). Cette variable n'a pas de valeur stockée connue (ATTENTION!!!!!!! selon les compilateurs elle peut contenir une valeur au hasard ou 0). La base mathématique utilisée pour représenter le nombre peut être différente de 10.

```
X : Long_integer := 15;       -- Notation Décimale
Y : Long_integer := 2#1111#;  -- Notation Binaire      (base  2)
```

```
W : Long_integer := 8#77#;    -- Notation Octale       (base  8)
Z : Long_integer := 16#F#;    -- Notation Hexadécimale (base 16)
```

Défini des variables dont le nom est X, Y et Z pour stocker des valeurs de -2^{64} à 2^{64} -1 (par exemple). X, Y, W et Z ont la même valeur 15. Observez bien la notation dans des bases autres que 10.

3.2.2.2 Range

On peut aussi définir un nouveau type d'entier, **non compatible directement** avec les deux types prédéfinis précédents.

```
Type my_integer_8 is range 1 .. 8;
```

Défini un ***NOUVEAU*** type contenant des entiers dont le nom est **my_integer_8** pour stocker des valeurs de –1 à 8.

Ce type est différent du type integer prédéfini par le compilateur!!!!!!

```
A_1: my_integer_8:= my_integer_8'first;
```

La variable A_1 est du type my_integer_8 **(entier différent de integer!!! car créé spécialement).** Elle a comme valeur 1 noter l'utilisation de l'attribut FIRST qui équivaut à la plus petite valeur possible.

3.2.2.3 Modulo

C'est un type d'entier sans signe avec une arithmétique appropriée (qui comprend les opérateurs logiques or, and, et xor bit à bit):

```
Type my_modulo is mod 4;
```

Les variables de ce type peuvent avoir les valeurs 0, 1, 2 et 3. Avec les particularités suivantes:
3 + 1 donne 0
0 – 1 donne 3
de même pour * et /

```
mod_4: my_modulo:= my_modulo'last;
```

Cette instruction définit une variable dont le nom est Mod_4 pour stocker des valeurs de 0 à 3. Cette variable a la valeur 3, noter l'utilisation de l'attribut Last qui équivaut à la plus grande valeur possible pour les variables de type **my_modulo**.

```
 Modular := 2 and 3;    -- modular est égal à 2
 Modular := 2 or  3;    -- modular est égal à 3
 Modular := 2 xor 3;    -- modular est égal à 1
 Modular := 3  +  1;    -- modular est égal à 0
```

l'utilisation d'un masque permet de travailler au niveau des bits.

```
 Modular := 2#10# and Mod_4 ;    -- modular est égal à 2
```

Les opérations suivantes sont utilisées de façon courante:

- Décalage de N bits à gauche.
  ```
  Modular := Modular * 2**N;
  ```
- Décalage de N bits à droite.
  ```
  Modular := Modular / 2**N;
  ```
- Effacement de bits avec le masque 02#0110#.
  ```
  Modular := Modular and 02#0110#;-- ici les bits 1 et 4 sont effacés (remplacés par des zéros).
  ```
- Mise à 1 de bits d'un mot avec le masque 02#0110#.
  ```
  Modular := Modular or 02#0110#;-- ici les bits 1 et 4 sont remplacés par des uns).
  ```
- Permutation circulaire des bits d'un mot de N fois à droite.
  ```
  Modular := (Modular *2**N) or (Modular /2**(Modular'size-N));-- Modular'size = 4 étant le nombre de bits de modular
  ```

L'utilisation de l'hexadécimal permet une lecture plus simple si le nombre de bits est élevé:

```
   Type  modulo_64 is mod 2**64 ; - donne une variable avec 64 bits soit 16 chiffres hexadécimal.

Z:     modulo_64;
.....
z:= z and 16#F000000000000000#;--ne garde que les 4 bits de poids le plus élevé de Z;

z := (z *2**N) or (z /2**(z'size-N));-- permutation circulaire
```

3.2.2.4 Positive

C'est un subtype (voir plus loin) **prédéfini** avec les limites suivantes (peuvent varier d'un compilateur à l'autre et aussi en fonction du **CPU**)
Positive varie de 1 à 2^{32} -1 (par exemple)

```
A_M_0: Positive := Positive'First;
```

Cette instruction définit une variable dont le nom est **A_M_0** pour stocker des valeurs de 1 à 2^{32} -1 (par exemple). Cette variable a la valeur 1, noter l'utilisation de l'attribut **First** qui équivaut à la plus petite valeur possible pour les variables de type **Positive**.

3.2.2.5 Natural

C'est un subtype (voir plus loin) **prédéfini** avec les limites suivantes (peuvent varier d'un compilateur à l'autre et aussi en fonction du **CPU**)
Natural varie de 0 à 2^{32} -1 (par exemple)

```
M_0: Natural := Natural'First;
```

Cette instruction définit une variable dont le nom est **M_0** pour stocker des valeurs de 1 à 2^{32} -1 (par exemple). Cette variable a la valeur 0, noter l'utilisation de l'attribut **First** qui équivaut à la plus petite valeur possible pour les variables de type **Natural**.

3.2.2.6 Les attributs applicables aux types entier ou dérivés

En plus des attributs **first** et **last** il existe aussi, entre autres, des attributs qui peuvent entre autre s'appliquer aux entiers :

```
Subtype test_type is integer range 1..5;
X: Test_Type:=2;
Y: Test_Type:=5;
Z: Test_Type;
```

Image

Transforme le contenu de la variable en un chaîne de caractères (principalement pour l'impression).
Test_type'image(x) renvoie le caractère 2 (avec un espace devant pour le signe +). Bien entendu, il existe dans le langage des méthodes permettant de contrôler le format des impressions.

Value

C'est l'inverse de la précédente, il transforme des caractères en un entier.

```
Z:= Test_type'Value(5);
```

Z contient maintenant 5.

Min

Permet de garder le minimum de deux variables

```
Z:= Test_type'Min(X,Y);
```

Z contient maintenant 2.

Max

Permet de garder le Maximum de deux variables

```
Z:= Test_type'Max(X,Y);
```

Z contient maintenant 5.

Range

C'est un attribut très utile qui équivaut à l'étendue des valeurs du type, il sert pour la logique, les tests et les boucles.

`2 in Test_type'range` renvoie vrai

3.2.3 Les options pour les types

Il est **peu pratique** (les types sont en effet **incompatibles entre eux voir 3.2.3.2**) de créer autant de types différents d'entiers que nécessaire pour couvrir les différences de nature des objets que l'on veux représenter. C'est pourquoi il existe un mécanisme pour limiter les bornes à partir d'un type (prédéfini ou créé).

3.2.3.1 Subtype

```
Subtype eee is my_integer_8 range 2 .. 5;
```

Cela donne un *type* dont le nom est eee compatible avec my_integer_8 mais limité à une plage de valeurs plus petite**.**

```
M_0: eee:= eee'last;
```

Cette instruction définit une variable de type **eee** dont le nom est **M_0** pour stocker des valeurs de 2 à 5. Cette variable a la valeur 5, noter l'utilisation de l'attribut **LAST** qui équivaut à la plus grande valeur possible pour les variables de type **eee**.

3.2.3.2 New

```
Type fruit_number_type is range 1..100;
```

[RAPPEL]
Cette instruction défini un ***NOUVEAU*** type contenant des entiers dont le nom est **fruit_number_type** pour stocker des valeurs de 1 à 100
Ce type est différent du type integer prédéfini par le compilateur!!!!!!
Vous ne pouvez pas les mélanger dans une même instruction, le compilateur vous donne un message semblable à:

```
xxx.adb:12:18: expected type "fruit_number_type" defined at line …
xxx.adb:12:18: found type "Standard.Integer"
gnatmake: "premier.adb" compilation error
```

Ce type peut servir à définir des type dérivés:

```
Type apple_number_type is new fruit_number_type;
Type grapefruit_number_type is new fruit_number_type;
```

Ces instructions définissent DEUX ***NOUVEAU*** types contenant des entiers dont les nom sont **apple_number_type** et **grapefruit_number_type** pour stocker des valeurs de 1 à 100. Ces deux types ***sont différent du type*** **fruit_number_type** mais ont ***conservé ses limites de valeurs de 1 à 100*****.**

```
apple_number: apple_number_type:= apple_number_type'first;
```

Cette instruction définit une variable de type **apple_number_type** dont le nom est **apple_number** pour stocker des valeurs de 1 à 100. Cette variable a la valeur 1, noter l'utilisation de l'attribut First qui équivaut à la plus petite valeur possible pour les variables de type **apple_number_type**.

```
grapefruit_number: grapefruit_number_type:= grapefruit_number_type;
```

Cette instruction définit une variable de type **grapefruit_number_type** dont le nom est **grapefruit_number** pour stocker des valeurs de 1 à 100. Cette variable n'a pas de valeur connue!!..

3.2.3.3 Private

Les types de variables définis dans un package (c'est une collection de fonctions et de procédures)
peuvent et sont utilisés dans les procédures et fonctions qui leur font référence dans la **ZONE 1** (voir plus haut). Le concepteur du programme DOIT décider de l'utilisation par l'extérieur des variables des types qu'il définit.

```
Type example  is private;
……
Private
……
Type example is integer range 1..2;
```

Les variables de ce type, déclarées à l'extérieur du «package» ne pourront **PAS** y être manipulées, seuls l'opérateur d'assignation (:=) est autorisé.

3.2.3.4 Limited private

Les types de variables définis dans un package (c'est une collection de fonctions et de procédures)
Peuvent et sont utilisés dans les procédures et fonction qui leur font référence dans la **ZONE 1** (voir plus haut). Le concepteur du programme DOIT décider de l'utilisation par l'extérieur des variables des types qu'il définit.

```
Type example  is limited private;
……
Private
……
Type example is integer range 1..2;
```

Les variables de ce type, déclarées à l'extérieur du «package» ne pourront PAS y être manipulées. L'utilisateur ne connaîtra pas les détails internes à ce type. Les objets de ce type sont en quelque sorte des mot de passe . Leur manipulation PASSERA OBLIGATOIREMENT par les procédures déclarées dans ce «package».

3.2.3.5 Aliased

```
type rr: aliased integer:=0;
```

Les variables de ce type pourront avoir leur adresse mémoire accessible par un pointeur.

3.2.4 Les réels

Les réels sont utilisée pour représenter les nombres avec décimales.

3.2.4.1 Digits

```
type Real is digits 2;
```

Cette instruction défini un ***NOUVEAU*** type contenant des réels dont le nom est **real** pour stocker des valeurs **avec deux chiffres après la virgule** la limite minimum et maximum ne sont pas précisées.
Le format de stockage des nombres dans la mémoire limite dans la réalité entre -3.40 10^{+38} et 3.40 10^{+38}.

```
type Coefficient is digits 10 range –1.0 .. 1.0;
```

Cette instruction défini un ***NOUVEAU*** type contenant des réels dont le nom est **real** pour stocker des valeurs **avec dix chiffres après la virgule**. la limite minimum et maximum est -1.0000000000 et 1.0000000000 .

```
AAA: Real;
BBB: Coefficient := Coefficient'last;
```

Ces instruction déclarent deux variables correspondant aux type ci-dessus:
AAA peut représenter des nombres réels à deux décimales et n'est pas initialisée.
BBB peut représenter des nombres réels à dix décimales, est initialisée à 1.0000000000 et est limitée entre –1 et 1.

3.2.4.2 Float ou long_float

Trois type prédéfinis **Float**, **Long_Float** et **Long_Long_Float** existent avec les spécifications suivantes

```
X: float := 0.0;
```

X peut varier entre -3.40282E+38 et 3.40282E+38 avec **7** chiffres significatifs (stockage mémoire sur 32 bits).

```
Y: long_float := 0.0;
```

X peut varier entre -1.79769313486232E+308 et 1.79769313486232E+308 avec **16** chiffres significatifs (stockage mémoire sur 64 bits).

3.2.4.3 Delta

```
type Volt is delta 0.125;
```

Cette instruction défini un type contenant des réels dont le nom est **volt** pour stocker des valeurs **arrondi à 0.125 prés** la limite minimum et maximum ne sont pas précisées.
Par exemple 1 1.125 1.250 1.275 etc. …. Ce peut être utilisé pour stocker des valeurs de mesure physique pour lesquelles la précision de mesure est connue (le voltmètre a une précision de 0.125 V dans cet exemple).

3.2.4.4 Delta et digits

```
type Money is delta 0.01 digits 15;
```

Cette instruction défini un type contenant des réels dont le nom est **Money** pour stocker des valeurs **arrondi à 0.01 près** la limite minimum et maximum ne sont pas précisées. Par exemple 1 1.125 1.250 1.275 etc. …. Ce peut être utilisé pour stocker des valeurs de physique pour lesquelles la précision de mesure est connue (le solde de votre compte en banque). 15 chiffres significatifs sont exigés pour les calcul (financiers dans cet exemple);

```
subtype Salary is Money digits 10;
```

C'est un exemple pour gérer les salaires des employés en modifiant une des caractéristiques.

3.2.4.5 Les attributs applicables aux types float ou dérivés

En plus des attributs applicables aux type entiers , il existe des attributs plus spécifiques:

3.2.5 Les booléens

C'est un type prédéfini qui peut avoir deux valeurs: **True** ou **False.**

```
D: Boolean:= False;
```

3.2.6 Les caractères

C'est un type prédéfini qui peut stocker un caractère;

```
A_char: character:= ' ';
```

A_char est initialisé avec un **ESPACE**!!!

Tous les caractères sont stockés en binaire par groupe de 8 bits selon un tableau. Ce tableau est prédéfini en ada . Par exemple:
A_char := Ascii.bs;

3.2.6.1 La table ASCII

Par exemple ^A signifie la clé ctrl et A en même temps

Dec	Hex	Char	clavier	Dec	Hex	Char	clavier	Dec	Hex	Char	Dec	Hex	Char
000	00	NUL	^@	018	12	DC2	^R	036	24	$	054	36	6
001	01	SOH	^A	019	13	DC3	^S	037	25	%	055	37	7
002	02	STX	^B	020	14	DC4	^T	038	26	&	056	38	8
003	03	ETX	^C	021	15	NAK	^U	039	27	'	057	39	9
004	04	EOT	^D	022	16	SYN	^V	040	28	(	058	3A	:
005	05	ENQ	^E	023	17	ETB	^W	041	29	)	059	3B	;
006	06	ACK	^F	024	18	CAN	^X	042	2A	*	060	3C	<
007	07	BEL	^G	025	19	EM	^Y	043	2B	+	061	3D	=
008	08	BS	^H	026	1A	SUB	^Z	044	2C	,	062	3E	>
009	09	HT	^I	027	1B	ESC	^[	045	2D	-	063	3F	?
010	0A	LF	^J	028	1C	FS	^\	046	2E	.	064	40	@
011	0B	VT	^K	029	1D	GS	^]	047	2F	/	065	41	A
012	0C	FF	^L	030	1E	RS	^^	048	30	0	066	42	B
013	0D	CR	^M	031	1F	US	^_	049	31	1	067	43	C
014	0E	SO	^N	032	20	SP		050	32	2	068	44	D
015	0F	SI	^O	033	21	!		051	33	3	069	45	E
016	10	DLE	^P	034	22	"		052	34	4	070	46	F
017	11	DC1	^Q	035	23	#		053	35	5	071	47	G

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
072	48	H	090	5A	Z	108	6C	l	126	7E	~
073	49	I	091	5B	[	109	6D	m	127	7F	
074	4A	J	092	5C	\	110	6E	n	128	80	~
075	4B	K	093	5D	]	111	6F	o	129	81	~
076	4C	L	094	5E	^	112	70	p	130	82	~
077	4D	M	095	5F	_	113	71	q	131	83	~
078	4E	N	096	60	`	114	72	r	132	84	~
079	4F	O	097	61	a	115	73	s	133	85	~
080	50	P	098	62	b	116	74	t	134	86	~
081	51	Q	099	63	c	117	75	u	135	87	~
082	52	R	100	64	d	118	76	v	136	88	~
083	53	S	101	65	e	119	77	w	137	89	~
084	54	T	102	66	f	120	78	x	138	8A	~
085	55	U	103	67	g	121	79	y	139	8B	~
086	56	V	104	68	h	122	7A	z	140	8C	~
087	57	W	105	69	i	123	7B	{	141	8D	~
088	58	X	106	6A	j	124	7C	|	142	8E	~
089	59	Y	107	6B	k	125	7D	}	143	8F	~

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
144	90	-	162	A2	¢	180	B4	Ž	198	C6	Æ
145	91	-	163	A3	£	181	B5	µ	199	C7	Ç
146	92	-	164	A4	€	182	B6	¶	200	C8	È
147	93	-	165	A5	¥	183	B7	·	201	C9	É
148	94	-	166	A6	Š	184	B8	ž	202	CA	Ê
149	95	-	167	A7	§	185	B9	¹	203	CB	Ë
150	96	-	168	A8	š	186	BA	º	204	CC	Ì
151	97	-	169	A9	©	187	BB	»	205	CD	Í
152	98	-	170	AA	ª	188	BC	Œ	206	CE	Î
153	99	-	171	AB	«	189	BD	œ	207	CF	Ï
154	9A	-	170	AA	ª	188	BC	Œ	206	CE	Î
153	99	-	172	AC	¬	190	BE	Ÿ	208	D0	Đ
155	9B	-	173	AD		191	BF	¿	209	D1	Ñ
156	9C	9	174	AE	®	192	C0	À	210	D2	Ò
157	9D	9	175	AF	¯	193	C1	Á	211	D3	Ó
158	9E	9	176	B0	°	194	C2	Â	212	D4	Ô
159	9F	9	177	B1	±	195	C3	Ã	213	D5	Õ
160	A0		178	B2	²	196	C4	Ä	214	D6	Ö
161	A1	¡	179	B3	³	197	C5	Å	215	D7	×

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
216	D8	Ø	234	EA	ê	252	FC	ü			
217	D9	Ù	235	EB	ë	253	FD	ý			
218	DA	Ú	236	EC	ì	254	FE	þ			
219	DB	Û	237	ED	í	255	FF				
220	DC	Ü	238	EE	î						
221	DD	Ý	239	EF	ï						
222	DE	Þ	240	F0	ð						
223	DF	ß	241	F1	ñ						
224	E0	à	242	F2	ò						
225	E1	á	243	F3	ó						
226	E2	â	244	F4	ô						
227	E3	ã	245	F5	õ						
228	E4	ä	246	F6	ö						
229	E5	å	247	F7	÷						
230	E6	æ	248	F8	ø						
231	E7	ç	249	F9	ù						
232	E8	è	250	FA	ú						
233	E9	é	251	FB	û						

3.2.7 Les énumérations

Il arrive souvent que l'on désire représenter dans un projet, des objets définis par leur nom. Par exemple: des fruits peuvent être de oranges, des pommes, des poires, des cerises, etc. Dans les langages anciens, il fallait coder ces espèces de fruits par des nombres: c'est peu pratique, sujet à des erreurs et ne respecte pas les principes de la programmation orientée objet. Ada propose un type spécial:

```
Type fruit_type is (orange, apple, pea, cherry);
A_fruit:fruit_type :=fruit_type'last ;
```

A_fruit est une variable pouvant avoir seulement les valeurs déclarées { orange ou apple ou …} dans le type et initialisé à cherry.

3.2.8 Les tableaux et matrices

Les tableaux (les matrices étant une généralisation des tableaux) permettent de stocker de nombreuses données de ***nature*** identique (de même type).
Chaque donnée individuelle est affectée d'un indice qui permet de les différencier de toutes les autres.
La notation traditionnelle s'écrit comme suit: X(i) ou x est le nom du tableau et i l'indice affecté à la valeur représenté par X(i).
Il faut IMPÉRATIVEMENT distinguer :

- Le type de variable contenu dans le tableau (la matrice).
- Le type de l'indice utilisé. Dans le cas des matrices les 2 indices peuvent être de nature différente !!

3.2.8.1 L'index et le contenu

Le type de variable contenu dans le tableau et **le type de l'indice utilisé** peuvent (ou non) être les mêmes. La limitation pour les indices est que leur nombre soit limité (pas de flottants par exemple).

L'opérateur de concaténation noté **&** permet de nombreuses opérations sur des fragments de tableaux. Il faut donc décider lors de l'analyse intellectuelle si ces possibilités seront utilisées (uniquement possible sur plusieurs tableaux de même type) car la déclaration pourra être différente ;
A cet effet il existe plusieurs méthode de déclaration de type de tableau. Il faut TOUJOURS commencer par déclarer les types de contenu et d'indice (sauf emploi des types prédéfinis par le langage).

3.2.8.2 Index de limites définies par des nombres

Un tableau

- contenant des entiers de 1 à 100
- indice entier de -2 à +3

```
Type U_Type is array (- 2 .. 3) of Integer range 1 .. 100;
 U : U_Type := (others => 1);
```

C'est la plus directe: le type de tableau est déclaré de façon explicite. Le tableau: variable de type tableau_type initialisé a 0 pour toutes les valeurs des indices{ -2, -1, 0, 1, 2 et 3}. L'inconvénient est qu'**aucun type ne correspond ni au contenu ni à l'indice**! Il n'est donc pas possible de déclarer des variables pour les représenter sans courir le danger de dépasser les limites des valeurs.

3.2.8.3 Index de limites définies par type

Un tableau

- contenant des booléens
- indice entier de -2 à +3

```
Type indice_type is new integer range –2..3;
Type tableau_type is array(indice_type'range) of boolean;

Tableau : tableau_type := (others=>true);
Indice  : indice_type;
Contenu : boolean;
```

On défini

- le contenu
- l'indice
- le type de tableau

On déclare le tableau: variable de type tableau_type initialisé à «true» pour toutes les valeurs des indices{ -2, -1, 0, 1, 2 et 3}.

Autre exemple:

Un tableau
- contenant des booléens
- indice fruit_type

```
Type fruit_type is (orange,apple,pear, cherry);
Type tableau_type is array(fruit_type'range) of boolean;

Tableau : tableau_type:=(others => true);
Indice  : indice_type;
Contenu : boolean;
```

Autre exemple
Un tableau
- contenant des fruit_type
- indice booléen

```
Type fruit_type is (orange,apple,pear, cherry);
Type tableau_type is array (boolean) of fruit_type;

Tableau : tableau_type:= (others => apple);
Indice  : boolean;
Contenu : fruit_type ;
```

3.2.8.4 Index de limites définies par les limites d'un subtype

Un tableau
- contenant des entiers de 1 à 2
- indice entier de -2 à +3

```
Subtype contenu_type is integer range 1..2;
Subtype indice_type is integer range -2..3;
Type tableau_type is array(indice_type'range) of contenu_type;
Tableau : tableau_type:=(others => 1);
Indice  : indice_type;
Contenu : contenu_type;
```

On défini le contenu, l'indice, le type de tableau , le tableau: variable de type tableau_type initialisé à 0 pour toutes les valeurs des indices{ -2, -1, 0, 1, 2 et 3}.
L'un des inconvénients est que la concaténation est limitée aux tableaux dont l'indice varie entre –2 et +3.

Autre exemple avec le même résultat
Un tableau
- contenant des entiers de 1 à 2
- indice entier de -2 à +3

```
Subtype contenu_type is integer range 1..2;
Subtype indice_type_a is integer range -2..3;
Subtype indice_type_b is integer range 10..20
Type tableau_type is array(integer range <>) of contenu_type;

Tableau_a : tableau_type(indice_type_a'range) := (others => 1);
Tableau_b : tableau_type(indice_type_b'range) := (others => 1);
```

Maintenant **tableau_a** **ET** **tableau_b** sont de MÊME **TYPE**.
De cette façon on pourra écrire:

```
 tableau_a(-2..0) := tableau_b(15..17);
```

Il y a aussi d'autres avantages que nous verrons plus loin.

3.2.8.5 Exemples de matrices

Dans la programmation avec des tableaux à N dimensions {matrices} il est particulièrement facile de mélanger lignes et colonnes si les deux indices sont de même type.

Les méthodes pour déclarer les tableaux se généralisent aux matrices.
Nous nous limiterons à quelques exemples:
Une matrice
- contenant des entiers de 1 à 200
- premier indice entier de -20 à +39
- second indice entier de 10 à +20

```
Subtype contenu_type is integer range 1..200;
type indice_type_a is NEW integer range -20..39;
```

```
type indice_type_b is NEW integer range 10..20;
Type matrice_type is array(integer range <>,
                           integer range <>) of contenu_type;

Matrice:matrice_type(indice_type_a'range,indice_type_b'range):=(others =>(others => 1));

indice_a : indice_type_a;
indice_b : indice_type_b;
```

La généralisation des écritures permet d'écrire

```
  x := matrice(indice_a,indice_b);
```

La ligne de code

```
  x := matrice(indice_b,indice_a);
```

Sera REJETÉE par le compilateur.

Une matrice

- contenant des fruits
- premier indice entier de -20 à +39
- second indice booléen
- initialisée à apple

```
Type fruit_type is (orange,apple,pear, cherry);
type indice_type is NEW integer range -20..39;
Type matrice_type is array(indice_type range <>,boolean range <>) of fruit_type;

Matrice: matrice_type(indice_type'range,boolean):= (others => (others => apple));
```

3.2.9 Les chaînes de caractères

Une chaîne de caractères est un **type prédéfini par le langage**: le type **string**. C'est un tableau de caractères avec un indice entier.
Sa définition POURRAIT s'écrire: array (positive range <>) of character;

Il possède toutes les caractéristiques des tableaux et en particulier, utilise l'opérateur &.
Toutes sortes de fonctions et procédures prédéfinies par le langage permettent de rechercher un mot, de transformer en minuscule, en majuscule, de traduire certaines lettres en d'autre lettres …….
Les bornes s'écrivent avec "

```
Chine: string (1..10);
R    : constant string :="bonjour"
V    : string (1..10)  :="01234456789";
-- attention!!! le compilateur compte les caractères il en faut 10

Subtype my_string_type is string(1..50);
my_string : my_string_type := (others => ' ');--initialisée avec 50 espaces
A_string  : my_string_type := (others => 'X');--initialisée avec 50 X
```

3.2.10 Les records

Décrivons un exemple de variables associées (par exemple appartenant au même objet). Le nom d'un article et son prix.

3.2.10.1 Les records simples

```
Type nom_type is (chaise,table,lampe,assiette);
Type prix_type is delta 0.10 digits 10 range 0.0 .. 1000.0;
Type article_type is record
Nom    : nom_type := nom_type'first;
Prix   : prix_type:= prix_type'first;
End record;

Article: article_type;
```

Le record 'capture' les propriétés de l'objet de l'exemple. La variable article a deux composants qui peuvent être d'un accès individuel:

```
article.Nom     := 10.51;
article.article := Lampe;
```

ou collectif:

```
article := (nom => table, prix => 20.0);
```

Bien que pratique dans les cas simples, il n'est pas adapté à des problèmes plus sophistiqués.

3.2.10.2 Les records avec discriminants

Il est possible d'inclure un tableau et sa taille dans un record:

```
type Storage_Array_Type is array (Long_Integer range <>) of integer;

type Storage_Type(Size : Long_Integer := Initial_Array_Size)
     is
     record
     Storage_Array         : Storage_Array_Type(0 .. Size);
     Current_Data_Position : Long_Integer := 0;
     Current_Last_Position : Long_Integer := 0;
     Bookmark_Position     : Long_Integer := 0;
   end record;


tableau: Storage_Type(Size => 10); -- initialisé avec un tableau d'indice 0 à 10;
```

3.2.10.3 Les records avec des parties variables et des choix discrets

Souvent les propriétés détaillées des articles sont différentes: il n'ont pas tous une couleur, se vendent au cageot, à la pièce ,….

```
Type nom_type is (chaise,table,pomme,poire,banane);
Type prix_type is delta 0.10 digits 10 range 0.0 .. 1000.0;
Subtype stock_type is integer range 1..100;
Type couleur_type is (blanc, brun, noir);

Type article_type (Article : nom_type := nom_type'first )
 is
 record
--c'est la partie commune à tous les articles :

stock: stock_type := stock_type'first;
```

-- partie spéciale pour table chaise et lampe qui sont compatibles : vente à l'unité et couleur

```
    case Article
       is
       when chaise | Table | Lampe=>
       prix_unitaire : prix_type    := prix_type'first;
       couleur       : couleur_type := couleur_type'first;

       when Pomme | poire =>
       prix_au_kilo        : prix_type := prix_type'first;
       prix_pour_un_cageot : prix_type := prix_type'first;

       when banane =>
       prix_a_la_piece     : prix_type := prix_type'first;
       prix_pour_un_regime : prix_type := prix_type'first;

End case;

End article_type;

Type banana_type is   article_type (Article=>banana);
Banana : banana_type;
.....
Banana. prix_pour_un_regime := …….;
```

Ici on distingue pour les différents articles la façon de les vendre: à l'unité, au kilo, par cageot ou par régime.
On se rapproche de la programmation 'objet' mais il manque l'héritabilité (entre autres).
On peut aussi, comme illustré plus haut définir un «subtype» limité au bananes.

3.2.11 Les pointeurs

L'un des concepts utilisés lors de la création du langage ADA a été de garder une représentation statique du problème aussi longtemps que possible. La conséquence est que l'analyse de la robustesse et de la validité du code peut être faite par le compilateur: si **un programme compile**, et si l'**analyse du problème a été bien faite** il **fonctionnera sans problème**. De plus, pendant son exécution, les idées incorporées dans le code seront testées (valeur des variables, indice de tableau compris dans les limites déclarées …). En effet certains langages concurrents, basés sur une utilisation préalable importante de l'assembleur utilise les pointeurs et l'arithmétique sur ceux–ci à un très grand degré. Cela même si ce n'est pas utile.

Les pointeurs en ada permettent un accès indirect à un «objet» en représentant l'adresse de cet objet qui peut être **une variable, une procédure, une fonction ou une tâche**. Il permettent, et c'est leur fonction la plus importante, la création dynamique d'objets n'existant pas quant le programme commence son exécution. Par exemple:

- Des objets peuvent être crées **et détruits** d'une manière imprévisible au moment de l'exécution du code (tampons dans un logiciel de messagerie, tâche opérant en parallèle, etc).

- **Plusieurs variables (de noms différents) peuvent représenter le même objet** (pour faciliter l'abstraction et la lecture du code selon le contexte envisagé).
- **Les relations entre les objets peuvent changer avec le temps** (les fils gauche et droits d'un arbre si on ajoute un composant ou on restructure l'arbre).
- **Stocker des données de nature différente dans la même structure** (entiers, réels et chaînes de caractères dans un même tableau).

Dés que l'objet n'est plus utile, il faut libérer la mémoire correspondante en utilisant l'instantiation d'un générique approprié !!!!
par exemple :

```
procedure Free is new Ada.Unchecked_Deallocation(Object => String,
                                                 Name   => String_access_type);
```

3.2.11.1 Déclaration de base

Créons un exemple de type de pointeur à partir du type prédéfini string:

```
Type String_access_type is access all string;
```

Cet exemple permet d'illustrer quelques emplois possibles, créons une variable ***qui est le pointeur***:

```
String_access: String _access_type := null;
```

Le mot clé **null** indique que le pointeur représenté par la variable **string_access** est vide et ne contient pas d'information valable. Le test d'un pointeur pour la valeur **null** est d'emploi courant (if null=string_access then ...).

```
Another_string_access : String _access_type:= new String'("bonjour");

--Ou alors
Another_string_access : String _access_type := null;
........
Another_string_access := new String'("bonjour");
```

C'est la création dynamique d'une chaîne de caractères, la variable Another_string_access pointe vers cette chaîne qui est accessible en utilisant l'opérateur de dé-référence **.all** :
Another_string_access.all est égal à "bonjour".
Ici, on ne défini pas la longueur de la chaîne de caractères à stocker, c'est un avantage primordial, le début et la fin sont accessibles par Another_string_access.all'first et Another_string_access.all'last:

```
Type String_array_type is arrray (1 .. 3) of String_access_type;

String_array : String_array_type := (1 => new'("01"),
                                     2 => new'("01234"),
                                     3 => new'("1"));
```

Cet autre exemple permet de construire **un tableau de chaînes de caractères de longueurs différentes**.

Il existe également la possibilité de **gérer le stockage associé** à ces pointeurs.

Par example:

```
with Ada.Unchecked_Deallocation;
…
…
procedure x …

Type String_access_type is access all string;
X : String_access_type := null;


procedure Free is new Ada.Unchecked_Deallocation(Object => String,
                                                 Name   => String_access_type);
…
Begin
……
X := new string'("ee");
……
Free (x);
……
```

3.2.11.2 Déclarations incomplètes

L'une des applications les plus intéressantes des pointeurs est l'utilisation de records dont un des éléments est **un pointeur sur ce même record**. Cela pose le problème de **l'œuf et de la poule:** Chaque déclaration de l'un des deux types (le pointeur ou le record) dépend de l'autre déclaration qui est à la ligne suivante et donc encore inconnue du compilateur. Inverser les deux lignes ne change rien au problème: la poule précède l'œuf ou l'œuf précède la poule la question n'est pas résolue!!.

Le langage ADA résous le problème en **ne précisant pas tout de suite la nature du type vers lequel pointeront les variables du type pointeur.**

```
type a_record;
```

La nature exacte de ce type n'est pas encore définie …..

```
type record_access is access a_record;
```

Cette déclaration est incomplète!

3.2.11.3 Terminaison d'une déclaration incomplète

```
type A_record
is
record
Value                     : Integer;
Pointeur_to_next_record   : record_access:=null;
end record;
```

La déclaration du record est complétée et contient un pointeur sur lui-même qui permettra **d'accrocher dynamiquement des nouveaux records** selon les besoins. Par exemple:

```
Record:A_Record;
 ..
Record := new A_Record'(Value                  => Data,
                        Pointeur_to_next_record => Record);
```

3.2.11.4 Pointeurs vers des objets composés (records et tableaux)

Examinons un exemple de déclaration:

```
type B is record
Plus : integer :=0;
Moins: integer :=0;
Egal : integer :=0;
end record;

type  Access_To_B is access  B;

X: Access_To_B;
```

Cela indique que tout nouvel objet de type Access_To_B créé par 'new B' sera initialisé tel que Plus, Moins et Egal seront à 0. L'allocation peut aussi avoir une initialisation explicite:

```
X:= new B'(Plus  => 2,
           Moins => 3,
           Egal  => 5);
```

Supposons l'utilisation d'un tableau dynamique tel que :

```
type Tableau_Type is array (integer range <>) of integer;
type Access_To_Tableau is access Tableau_Type;
```

pendant la création aucune borne n'est nécessaire:

```
V:  Access_To_Tableau;
W:  Access_To_Tableau;
```

Au moment de stocker des valeurs, la longueur doit être précisée:

```
V:= new  Tableau_Type(3..19);

V:= new  Tableau_Type(1..7);
```

le dé-référencement s'écrit alors

```
V.all(i);
X.Plus.all
```

3.2.11.5 Assignation et test avec pointeurs

Attention!! les assignations n'ont pas le même sens et ne conduisent pas au résultat à priori évident:
les valeurs deviennent identiques
```
X.all:=Y.all;
```
et
les pointeurs vont au même endroit
```
X:=Y;
```

sont différentes!!!

Les test peuvent s'écrivent:

```
if X = null --  aucun objet
then
.....
end if;

if X.all = Y.all  -- valeurs pointées  égales
then
.....
end if;
```

3.2.11.6 Effacement d'un pointeur et des données associées

Quant le pointeur et son stockage associé ne sont plus nécessaires, il faut restituer la mémoire correspondante pour éviter les "fuites de mémoire". Par exemple:
déclaration du pointeur
```
type access_to_string is access all string;
```
instantiation de la procédure de restitution de la mémoire correspondante
```
procedure Free_Node is new Ada.Unchecked_Deallocation(Object => string,
                                                      Name   => access_to_string);
```
déclaration du pointeur
```
A: access_to_string;
```
assignation du contenu du stockage associé
```
A:=new string'("abc");
```
restitution de la mémoire correspondante
```
Free_Node(A);
```
assignation à une nouvelle valeur
```
A:=new string'("cde");
```

3.2.12 Programmation objet

Il n'est pas possible d'écrire de grandes applications sans utiliser les techniques modernes de programmation. Nous allons prendre l'exemple du magasin qui vend:

Des tables, des chaises, des oranges et des bananes.

Plusieurs opérations s'appliquent à ces objets (au sens propre et programme) vendus dans ce magasin. Mais, leurs différences sont suffisante (durée de stockage, unité de vente, unité de stockage ….) pour justifier un traitement différent:

- La durée de stockage ne s'applique pas aux tables et chaises
- L'unité de vente peut être au kilo pour les oranges et les bananes
- Une option d'unité de vente: au cageot pour les oranges, au régime pour les bananes avec des prix différents du prix au kilo;
- La durée maximum de stockage est très différente entre les oranges (durée longue) et les bananes (durée courte).
- Le délai de commande peut être significatif pour l'ameublement mais n'a pas de sens pour les fruits.

3.2.13 Abstract

Propriétés communes à tous ces objets:
Ils sont vendus dans le magasin et c'est tout !! on capture cette analyse (on dit abstraction) par une déclaration:

```
Type objet_vendus_dans_le magasin_type is abstract tagged with private;
```

L'utilisation du mot clé **abstract** indique que à ce stage, je ne désire pas de possibilité de traitement des données. Il sont vendus dans le magasin, ce qui peut se traduire par:

```
Procedure vendre(quoi: in out objet_vendus_dans_le magasin_type) is abstract;
```

A nouveau, a ce stade aucune action avec le mot clé **abstract**.

Private a son sens habituel de restriction d'usage.

3.2.14 Tagged

Bien sûr, je vais différencier par étapes successive entre les différents objets qui vont hériter des propriétés communes introduites dans les étapes précédentes .

3.2.15 Extension et héritabilité des types "tagged"

Tous ces objets sont stockés par unité de stockage (cageot, régime ou à la pièce)

```
Type unite_de stockage_type is (cageot, regime, par_piece);

Type objets_en_stock_type is new objet_vendus_dans_le magasin_type
With record
Unite_de_stokage: Unite_de_stokage_type:= Unite_de_stokage_type'first;
End record;
```

Et ainsi de suite pour les autres propriétés.

3.2.16 Le type task (tâche)

Ce type permet la création et l'exécution en parallèle de plusieurs parties du projet de manière indépendante et simultanée. Il possède de nombreuses possibilités telle que rendez-vous, délais variés.
L'accès simultané de plusieurs tâches aux mêmes données présente la possibilité d'erreurs. Ada a prévu ce problème grâce à un «type particulier de package» :

```
protected type Protected_Storage_Type
     is
…..
end Protected_Storage_Type;
```

Toutes ces possibilités seront discutées dans la suite de ce cours.

3.3 DOMAINE D'EXISTENCE DES VARIABLES

Il est très important de prévoir expressément :

- Le domaine d'existence pour chaque variable utilisée dans un programme
- Le domaine de visibilité des opérations pour chaque variable utilisée dans un programme.

Ces domaines peuvent être très variés:

3.3.1 Domaine d'existence limité à quelques lignes dans un code

C'est le cas, par exemple, pour l'indice d'une boucle for:

```
Dummy existe uniquement dans l'intervalle  loop .. end loop
...
...
For Dummy in 1..Er'Last loop
• • •
z:=2*I;
• • •
end loop;
...
```

L'utilisation de "declare" permet aussi une limitation du domaine d'existence:

```
procedure r
is
....
begin
....
....
x existe entre sa déclaration et end
declare
X:Integer;
begin
X:=Z+X;
end;
...
...
....
end r;
```

```
...
```

3.3.2 Domaine d'existence limité à toutes les lignes d'une procédure / fonction

C'est le plus classique:

```
....
....
procedure z
is
x:integer;
y existe entre sa déclaration et end z
y:integer;
begin
....
....
y:=y+1;
....
end z;
....
...
```

3.3.3 Domaine d'existence limité à une partie des procédures / fonctions d'un package

Aucune garantie n'est donnée pour la conservation de la valeur stockée entre deux appels à une procédure / fonction du package (en général elle est conservée).

```
Package body e
is

procedure One (...)
is
begin
...

end one;
Str  "existe" entre sa déclaration et end E
str: string (1..10);

procedure two (...)
is
begin
...

end two;
...
end E;
```

3.3.4 Domaine d'existence limité à toutes les lignes d'un package

Il y a plusieurs cas possibles:

3.3.4.1 Le domaine est limité à toutes les lignes du package avec impossibilité d'accès externe

3.3.4.1.1 Déclaration dans le "body"

Aucune garantie n'est donnée pour la conservation de la valeur stockée entre deux appels à une procédure / fonction du package (en général elle est conservée).

```
Package body e
is
Str  "existe" entre sa déclaration et end E
str: string (1..10);

procedure One (...)
is
begin
...

end one;
```

```
procedure two (...)
is
begin
...

end two;
...
end E;
```

3.3.4.1.2 Déclaration dans la spécification

La conservation de la valeur (str) stockée entre deux appels à une procédure / fonction du package est garantie

Fichier e.ads (spécification):

```
Package   e
is

procedure One (...) ;
procedure two (...) ;
après private aucune vue externe des variables
private
str: string (1..10);
...
end E;
```

Fichier e.adb (body):

```
Package body e
is
procedure One (...)
is
begin
...
end one;
...
procedure two (...)
is
begin
...
end two;
...
end E;
```

3.3.4.2 Le domaine est limité à toutes les lignes du package avec possibilité d'accès externe

Dans ce cas la conservation de la valeur stockée entre deux appels à une procédure / fonction du 'package' est garantie.

Fichier e.ads (spécification):

```
Package   e
is

type xyz_type is ........;
xyz: xyz_type;
...
procedure One (...) ;
procedure two (...) ;

end E;
```

Fichier e.adb (body):

```
Package body e
is
procedure One (...)
is
begin
...
end one;
...
procedure two (...)
is
begin
...
end two;
...
end E;
```

3.3.5 Domaine d'existence limité à toutes les lignes de tous les packages en utilisant "with ..."

C'est une variable globale, chaque fois que "with e;" est présent cette variable est globale pour ce package. Attention: "with e;" n'importe pas les opérateurs associés à cette variable pour cela il faut mettre "use type e.xyz_type;"

```
with e;
Package body z
is
procedure One (...)
is
Accès aux opérateurs associés au type  e.xyz_type
use type  e.xyz_type;
begin
Accès à une variable globale associée au package e
my_xyz: e.xyz_type;
...
end one;
...
procedure two (...)
is
begin
...
end two;
...
end E;
```

3.3.6 Utilisation de "finalize" pour illustrer le domaine d'existence

Le langage ada à prévu la possibilité d'action à la création et à la destruction d'une variable: voici un exemple simpliste:

Le programme bien que vide donne le résultat suivant:

[dg@newdg miage_2005_2006]$ ./z
Initialize
Finalize

Fichier x.ads:

```
with Ada.Finalization;
package X
is
Le type example est vide dans ce cas simpliste mais ce n'est pas l'emploi le plus fréquent.

   type Example
      is new Ada.Finalization.Limited_Controlled
      with null record;

   procedure Initialize(Object : in out Example);

   procedure Finalize(Object   : in out Example);

end X;
```

Fichier x.adb:

```
with Ada.Text_Io;
with Ada.Finalization;

package body X
   is
procédure appelée lors de la création
   procedure Initialize(Object : in out Example)
      is
      begin
      Ada.Text_Io.Put_Line("Initialize");
   end Initialize;
procédure appelée lors de la destruction
   procedure Finalize(Object : in out Example)
      is
      begin
      Ada.Text_Io.Put_Line("Finalize");
   end Finalize;
end X;
```

Fichier z.adb:

```
with X;
procedure Z
   is
création de la variable, appel automatique de Initialize
```

```
   Dummy : X.Example;
   begin
   null;
destruction de la variable: appel automatique de Finalize
end Z;
```

4 LE DÉCOUPAGE DES ROUTINES EXTERNES EN FICHIER 'ADS' ET 'ADB'

Tout projet, **même simple** DOIT ÊTRE DÉCOUPÉ EN PLUSIEURS ROUTINES. L'exemple le plus simple est:

- Une routine de définition des types de variables.
- une routine entrée/sortie.
- une routine de calcul.

les fichiers se distinguent par leur nom, leur extension et le début de leur contenu. Tout projet ada contient un programme principal qui est une procédure sans paramètres. Le nom du fichier stocké est celui du nom de la procédure sans paramètre (cela oblige le nom du fichier à être un nom de procédure valide en ada). Par exemple, le fichier a.adb contiendra

```
--  des commentaires
with ……..;
 procedure a
is
……
begin
……
end a;
```

4.1 LE CHOIX ENTRE UNE FONCTION ET UNE PROCÉDURE

Il existe 2 choix principaux pour découper un problème en routines indépendantes: la procédure et la fonction. Le choix se fait selon plusieurs critères définis ci-dessous.

4.1.1 Le passage de paramètres

Le passage de paramètres (variables, tableaux, pointeurs,…….) se fait selon trois modes différents.
Ces modes permettent de 'capturer votre analyse intellectuelle' du problème et d'assurer un programme conforme, écrit plus rapidement et avec moins d'erreurs.

4.1.1.1 Le mode in

Dans ce mode, la 'variable' sera en **lecture** seule dans cette routine (procédure ou fonction). C'est le mode par défaut si rien n'est précisé.

```
procedure Test(I : in  Integer := 2;
               A : in  Integer;
               B :     Integer)
```

Is
begin
…..
end Test;

Remarquez la possibilité d'introduire une valeur par défaut pour la variable I. Toutes les variables I, A et B sont disponibles dans cette procédure avec les valeurs transférées lors de l'appel (pas de déclaration entre is et begin!!!). Par exemple, les trois instructions suivantes sont équivalentes:

```
Test(I => 1,
     A => 2,
     B => 3);

Test(1,
     2,
     3);

Test(1,
     A => 2,
     B => 3);
```

Elles appellent la procédure TEST avec les valeurs 1 pour I, 2 pour A et 3 pour B.

Les instructions suivantes appellent la procédure en utilisant la valeur par défaut.

```
Test(A => 2,
     B => 3);
```

```
Test(2,
     B => 3);

Test(2, 3);
```

4.1.1.2 Le mode out

Dans ce mode, la 'variable' sera en **écriture** seule (lecture permise après une écriture) dans cette routine (procédure ou fonction).

```
procedure Test(I : in  Integer := 2;
               A : out Integer;
               B : out Integer)
Is
…..
end Test;
```

Les appels ont la même syntaxe que pour le mode in.

4.1.1.3 Le mode in out

Dans ce mode, la 'variable' sera en **lecture écriture** dans cette routine (procédure ou fonction)

```
procedure Test(I : in     Integer := 2;
               A : in out Integer;
               B :    out Integer)
```

Is
…..
end Test;

Les appels ont la même syntaxe que pour le mode in.

4.1.1.4 Access

Dans cette possibilité la 'variable' sera un pointeur en lecture seule.

```
procedure Test(I : in     Integer := 2;
               A : in out Integer;
               B : Access Object'Class)
```

Is
…..
end Test;

4.1.2 Les fonctions

La fonction prend plusieurs arguments (in par défaut), **tous en lecture seule**, et renvoie **UNE SEULE VALEUR**.
Elle s'apparente beaucoup à une variable dans sa syntaxe d'utilisation.

```
 function A(A : Integer;
            B : Float)
   return Float
   is
   begin
   return B**A;
 end A;
```

Cette fonction très simple renvoie B puissance A. Notez la syntaxe.

La syntaxe d'appel est simple, il suffit de se souvenir que c'est la même utilisation qu'une variable.
Voici trois exemples équivalents (my_float est un réel déclaré avant):

```
My_Float:=A(2,3.0);

My_Float:=A(2,
B => 3.0);

My_Float:=A(A => 2,
            B => 3.0);
```

4.1.3 Les procédures

Une procédure peut ne renvoyer aucune donnée (pas de paramètre, ou tous en mode **in**) mais ce n'est pas le cas général. Les paramètres en mode **in** peuvent avoir une valeur par défaut dans ce cas le paramètre peut être omis dans l'appel.

```
procedure A(I : in     Integer := 2;
            A : in out Integer;
            B :    out Integer)
  is
  begin
  B := A / I * I;
  A := A mod I;
end A;
```

La syntaxe d'appel à été explicitée auparavant. Voici deux exemples:

Utilisation de la valeur par défaut pour I

```
A(A =>1,
  B =>2);
```

Appel normal sans utilisation de la valeur par défaut

```
A(I => 4,
  A =>1,
  B =>2);
```

5 LA SYNTAXE DU CODE

5.1 LA SYNTAXE DES AFFECTATIONS

L'opérateur := est utilisé pour l'affectation du résultat à une variable d'un type précis. La syntaxe dépend du type, voici quelques exemples:

```
x :=     4.0;
x :=     Pi;
x :=     (1 .. 10 => 0);
x :=     Sum;
x :=     indice_type_a'Last;
x :=     Sine(X);
x :=     long_integer'(2);
x :=     Float(M * N);
x :=     Z * (X + 10);
```

```
x :=     Volume;
x := not Destroyed ;
x := 2 * Line_Count;
x := - 4.0;
x := - 4.0 + A;
x := B ** 2 - 4.0 * A * C;
x := Password(1 .. 3) = "Bwv";
x := Count in indice_type_a;
x := Count not in indice_type_a;
x := Index = 0 or over;
x := (Rouge and vert) or jaune;    -- (parenthèses obligatoires)
x := A ** (B ** C);                -- (parenthèses obligatoires)
x := (1      => 1,
      others => 2);
```

```
x:= (un           => 1,
     deux | trois  => 2,
     fin          => 35);
x.un   := 1;
x.deux := 2;
x.trois:= 2;
x.fin  := 35;
x      := T(1..N) & S(3..5);

x:= Ma_Fonction(A => 2.0,
                B => 5.0);

X:= new my_type'(x);

X := new Tagged_Data_Type'Class'(The_Record);

X := new Node'(Previous => null,
               The_Item => From_Index.The_Item,
               Next     => null);

Variable : C.My_Type := C.My_Type'(Ada.Finalization.Controlled with I => 0);
```

5.1.1 Opérateurs logiques

Les opérateurs classiques and or xor sont définis comme d'habitude, ils s'appliquent aussi A certains entiers.

A	B	(A and B)	(A or B)	(A xor B)
True	True	True	True	False
True	False	False	True	True
False	True	False	True	True
False	False	False	False	False

OR ELSE ou alors: le second terme n'est évalué qui si le premier terme est faux.
AND THEN et alors : **le second terme n'est évalué qui si le premier terme est vrai.**
IN compris dans l'intervalle
.. intervalle

Examples:

```
Rouge or jaune

A(1 .. 10) and B(15 .. 24)

pointeur /= null and then Age > 25

N = 0 or else X(N) = vert

z in 1..5    -- z compris entre 1 et 5
```

un petit programme:

```
with Ada.Text_Io;

 procedure One
 is

    type Color_Type is(Yellow, Blue, Red, Cyan, Green, Magenta);

    Color : Color_Type;
    Test  : Boolean;

 begin

    Color := Red;
    Test  := Color = Yellow or Color = Red;
    Ada.Text_Io.Put("test is " & Boolean'Image(Test));

 end One;
```

et sa sortie écran:

```
test is true
```

5.1.2 Opérateurs de comparaison

Ces opérateurs classiques sont définis comme d'habitude:

= égal
/= différent

< inférieur
<= inférieur ou égal
> supérieur à
>= supérieur ou égal

un exemple simple:

```
with Ada.Text_Io;

   procedure Two
   is

      A : Integer;
      B : Integer;
   begin
      A := 2;
      B := 3;
      Ada.Text_Io.Put(" 2 > 3 is " & Boolean'Image(2 > 3));


   end Two;
```

Et sa sortie écran:

```
 2 > 3 is true
```

5.1.3 Opérateurs d'addition

Ces opérateurs classiques sont définis comme d'habitude. L'opérateur & s'applique seulement au tableaux (dont les chaînes de caractères) et correspond au +.
+ plus
– moins
& concaténation
| ou dans les 'case' et les assignations

```
x:= (un           => 1,
     deux | trois  => 2,
     fin          => 35);

case x is
when 1|2 => ....
....
end case;
```

5.1.4 Opérateurs de signe

Rien de spécial.
+ –

5.1.5 Opérateurs de multiplication

Ces opérateurs classiques sont définis comme d'habitude.

* Multiplication
/ Division
mod Modulo
rem Reste de la division

A	B	A/B	A rem B	A mod B	A	B	A/B	A rem B	A mod B
10	5	2	0	0	–10	5	–2	0	0
11	5	2	1	1	–11	5	–2	–1	4
12	5	2	2	2	–12	5	–2	–2	3
13	5	2	3	3	–13	5	–2	–3	2
14	5	2	4	4	–14	5	–2	–4	1

A	B	A/B	A rem B	A mod B	A	B	A/B	A rem B	A mod B
10	–5	–2	0	0	–10	–5	2	0	0
11	–5	–2	1	–4	–11	–5	2	–1	–1
12	–5	–2	2	–3	–12	–5	2	–2	–2
13	–5	–2	3	–2	–13	–5	2	–3	–3
14	–5	–2	4	–1	–14	–5	2	–4	–4

5.1.6 Opérateurs de rang le plus haut

Ces opérateurs classiques sont définis comme d'habitude.

**	Élévation à une puissance
abs	Valeur absolue
not	Négation logique (voir plus haut)

5.1.7 Modification du sens des opérateurs classiques

Il est toujours possible, pour permettre une lecture plus facile d'un code, de renommer les opérateurs (ou les fonctions et procédures).

```
Function mult_plus_un (left  : in integer;
                       right : in integer )
return integer
is
begin
return left*right+1;
end   mult_plus_un;
.....
function "*" (left :in integer; right : in integer ) return integer  renames  mult_plus_un;
....
....
begin
x:= a*b; --  calcule a*b+1
...
```

5.1.8 Conversion entre types

Bien que peu utile, Il faut savoir l'utiliser

5.1.8.1 Entier vers réel

```
X := float (i);
```

5.1.8.2 Réel vers entier

```
I := integer(x);
```

5.1.8.3 Conversion par utilisation d'une adresse mémoire commune

Cette méthode permet l'accès à un niveau très bas. Elle permet de lire la même adresse mémoire de deux façons différentes si les deux type sont représentés par le même nombre de bits. Son utilisation, un peu complexe, est illustrée dans les tables de hashing.

5.2 LA SYNTAXE DES TESTS

Le langage permet deux types différents: if et case.

5.2.1 Le test if

La grammaire ada indique:

if condition then sequence_of_statements
{elsif condition then sequence_of_statements}
[else sequence_of_statements]
end if;

Les abréviations sont les suivantes:
{ xxxxxx **}** veut dire 0 à n répétitions de xxxxxx
[yyyyyyy **]** veut dire yyyyyy est optionnel
sequence_of_statements veut dire 1 à N instructions

exemples :

```
If my_test = 1
Then
X:=I;
End if;


If my_test = 1
Then
X:=I;
……
Elsif my_test = 0
Or else My_Test = 5
Then
Y:=I;
……
Elsif my_test in 2..4
Then
W:=I;
……
else -- optionnel!!
z:=I;
…
end if;
```

5.2.2 Le test case

Le test case permet beaucoup plus de possibilités et est plus lisible s'il y a beaucoup de 'elsif';
il oblige à traiter tous les cas, si tous les cas ne sont pas pris en compte on termine par others.
Pour le "else" final
Voici le même code que ci-dessus traduit en case:

```
Case My_Test
Is
when 1 =>
X:=I;
……
when  0 | 5 =>  -- 0 ou 5
Y:=I;
……
when  2 .. 4 => -- 2, 3 et 4
W:=I;
……
when others =>   -- doit être le dernier choix
z:=I;
…
end case;
```

voici un autre exemple

```
Type jour_type is (lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche);
Jour: Jour_type;
…
Begin
……
case jour
is
when lundi => nouvelle_semaine;
when  mardi .. jeudi =>  travail;
when vendredi => vivement_ce_soir;
when samedi | dimanche  =>     -- peut s'écrire samedi .. dimanche
repos;
end case;
```

5.3 LA SYNTAXE DES BOUCLES

Les boucles sont des outils très importants dans la programmation. La sortie de boucle s'effectue:
Boucle «**while**» : condition de «while» fausse
Boucle «**for**» : indice supérieur à la borne
Toutes boucles: exit simple ou exit conditionnel

5.3.1 Boucle simple

```
Nom_de_boucle :   -- optionnel mais utile si  plusieurs boucles imbriquées
```

```
Loop

……
-- boucle infinie si pas de sortie
exit nom_de_boucle when a=0; -- exit conditionnel  nom de boucle optionnel
exit nom_de_boucle;          -- exit simple        nom de boucle optionnel

……
end loop Nom_de_boucle;
```

5.3.2 Boucle while

```
While a>0 or else r <1 loop
  ……
End loop;
```

La boucle se répète tant que la condition a>0 or else r <1 est vraie.

5.3.3 Boucle for

Ce type de boucle est souvent utilisé pour les indices de tableaux ou pour répéter le même code plusieurs fois.

```
Subtype contenu_type is integer range 1..2;
Subtype indice_type_a is integer range -2..3;
Subtype indice_type_b is integer range 10..20;
Type tableau_type is array(integer range <>) of contenu_type;
Tableau_a: tableau_type(indice_type_a'range):=(others=>1);
Tableau_b: tableau_type(indice_type_b'range):=(others=>1);
…
begin
…
x := 0;
for I in 1..2 loop
x:=x + Tableau_a(I);
end loop;

x := 0;
for I in tableau_a'range loop
x:= x + Tableau_a(I);
end loop;

nom_de_boucle :
for I in  reverse tableau_a'range loop
for J in tableau_b'range loop
x:= x + Tableau_a(I)* tableau_b(j);
exit nom_de_boucle when 0 = Tableau_a(I);
end loop;
end loop_nom_de boucle;
```

I et J ne sont PLUS disponibles après la boucle. Si cela est nécessaire, il faut utiliser une boucle while, et modifier les valeurs des indices par vous même.

```
……
I: integer ;
J : integer ;
……
Begin
……
I:= indice_type_a'last;
J:= indice_type_b'first;

nom_de_boucle :

while I in tableau_a'range loop
while J in tableau_b'range loop

x:= x + Tableau_a(I)* tableau_b(j);
exit nom_de_boucle when 0 = Tableau_a(I);
J:=1 + J;
end loop;

I:= I - 1;
end loop_nom_de boucle;
```

5.4 LA SYNTAXE D'APPEL DES FONCTIONS

Il suffit de se souvenir qu'une fonction s'utilise comme une variable.
Par exemple, une fonction de spécification:

```
Function abc(premier: in integer:=1;
             Second : in integer) return float;

Peut s'appeler comme suit:

A: integer;
```

```
B : integer;
C: float;
...
...
--premier prends la valeur de a et second la valeur de b
c := 2.0 * Abc(a,b);
--premier prends la valeur de a et second la valeur de b

c:= abc(premier => a,
        second  => b);
--premier prends la valeur  par défaut de 1 et second la valeur de b

c:= c + abc(second  => b);
```

5.5 LA SYNTAXE D'APPEL DES PROCÉDURES

Soit une procédure:

```
Procedure  Prod (A_1: in out A_type;
                 A_2 : in     A_Type;
                 A_3 :  out   A_type :=a_type'first);
```

L'appel pourrait se faire comme suit

```
X: A_Type;
Y: A_Type;
Z: A_Type;
....
.....
.....
prod(x,y,z);

prod(x,y);-- utilisation de la valeur par default

prod (A_1=> 2 * X - 1,
      A_2 =>  X * y,
      A_3 =>  X * Y / z);
```

5.6 LA SEGMENTATION EN PARTIES INDÉPENDANTES PAR 'DECLARE' OU BEGIN

Le programme peut (***et doit***) être séparé en sections indépendantes. Il en existe toujours une par défaut entre begin et end. Toute erreur pendant l'exécution à l'intérieur de ces parties **DOIT** être gérée, voir plus bas. Des variables peuvent ***éventuellement*** être déclarées dans un nouvelle section (qui peut avoir un nom) et n'être valable que dans cette section. **Cette méthode permet de déclarer des tableaux (matrices) dynamiquement: leur longueur n'est pas connue du compilateur mais déterminée pendant le fonctionnement du programme**:

```
Procedure X …..
is
size :integer;
type table_type is array (integer range <>) of integer;
…
begin
…    lecture de size ou affectation
size :=..........;

declare    -- nécessaire si variable à déclarer
x: integer;
table: table_type(1..size);
begin – début d'une section supplémentaire
x:= .....;
…
table(x):=........;

end; -- fin de cette section supplémentaire
-- X et table n'existent plus !!!!

endX;
```

Le même exemple avec un nom de section

```
Procedure X …..
is
…
begin
…
interne :  -- nom de la section
declare    -- nécessaire si variable à déclarer
x: integer;
begin – début d'une section supplémentaire
x:= .....;
…
```

```
end  interne; -- fin de cette section supplémentaire
-- X n'existe plus !!!!

endX;
```

5.7 GESTION DES ERREURS DURANT L'EXÉCUTION DU PROGRAMME

Il faut toujours programmer en prenant en compte **que l'utilisateur peut et fera des erreurs**!
Pour le gérer on utilise la notion d'exceptions. C'est une réponse à une situation anormale.
…

5.7.1 La notion d'exception

Par exemple la racine carrée d'un nombre négatif n'est pas définie (sans nombres imaginaires!). On déclarera donc une exception:

```
Racine_imaginaire: exception;
```

Elle pourra servir plus loin dans le code par exemple

```
If x <=0.0
Then
Raise Racine_imaginaire;-- cette instruction renvoie à la gestion d'erreur et va
                        -- directement dans exception à la ligne correspondante
…..
end if;
.......
......
exception
when  Racine_imaginaire =>
...... traitement de l'exception
when ....=>
........
when others =>
... traitement des  exceptions non prise en compte ci dessus
 end ....; fin de la routines
```

5.7.2 Les exception intégrées au langage

Les fonctions déjà définies dans le langage ont très souvent des exceptions intégrées.

5.7.3 Définition d'une exception

On utilise le mot réservé exception:

```
Table_pleine, pointeur_null: exception;
```

5.7.4 Gestion des exceptions

Elle se fait à la fin d'un block défini entre begin et end après la dernière instruction et avant end.

```
…..
begin
….
If x <=0.0
Then
Raise Racine_imaginaire;-- cette instruction renvoie après exception
else
X:= racine-carrée(y);
…
end if;

exception
when  Racine_imaginaire =>
… -- ici on gère les conséquences de l'erreur
when .... => .....

end;
```

5.7.5 Impression écran du type d'erreur rencontré

Si on désire imprimer le nom de l'erreur (avec le nom de la procédure):

```
with Ada.Exceptions;
with Ada.Text_Io;
....
procedure  Initialize_Generator(......)
.....
begin
....
     exception

      when Programing_Error : others =>
      Ada.Text_Io.Put("Initialize_Generator"
                      & Ada.Exceptions.Exception_Information(Programing_Error));
   end Initialize_Generator;
....
```

6 ENTRÉES ET SORTIES

6.1 ENTRÉE CLAVIER

Principes généraux:

Il y a deux choix possibles: cacher tous les détails dans un package spécial (forcément limité dans ses possibilités voir le «package ou tout expliquer en détail au risque de paraître fastidieux. Des exemples tout prêts seront utilisés.

6.2 UTILISATION DE READ_WRITE

Ce programme déjà écrit pour vous vous permet de ne pas entrer dans les détails du fonctionnement des entrées sorties.
Voici les possibilités offertes ainsi qu'un exemple pour chaque

6.2.1 Entrée d'un entier

Vous pouvez aussi utiliser le code de Read_Write. (Voir Chapitre 14.1)

```
with Read_Write;

procedure XX
is
x:Integer;
begin
....

Read_Write.Read(Header => "valeur de x ? " ,
                Value  => X);
....
end XX;
```

6.2.2 Sortie écran d'un entier

```
with Read_Write;

procedure XX
is
x:Integer;
begin
....

Read_Write.Write(Header => "La valeur de x  est ",
                 Value  => x);
....
end XX;
```

6.2.3 Entrée d'un entier long

```
with Read_Write;

procedure XX
is
x:Long_Integer;
begin
```

```
....

Read_Write.Read(Header => "valeur de x" ,
                Value  => x);
....
end XX;
```

6.2.4 Sortie écran d'un entier long

```
with Read_Write;

procedure XX
is
x:Long_Integer;
begin
....

Read_Write.Write(Header => "valeur de x" ,
                 Value  => x);
....
end XX;
```

6.2.5 Entrée d'un réel

```
with Read_Write;

procedure XX
is
x:Float;
begin
....

Read_Write.Read(Header => "valeur de x" ,
                Value  => x);
....
end XX;
```

6.2.6 Sortie écran d'un réel

```
with Read_Write

procedure XX
is
x:Float;
begin
....
read_write.Write("valeur de x" , x);
....
end XX;
```

6.2.7 Entrée d'un réel long

```
with Read_Write;

procedure XX
is
x:Long_Float;
begin
....

Read_Write.Read(Header => "valeur de x" ,
                Value  => x);
....
end XX;
```

6.2.8 Sortie écran d'un réel long

```
with Read_Write

procedure XX
is
x:Long_Float;
begin
....
read_write.Write("valeur de x" , x);
....
end XX;
```

6.2.9 Entrée d'une chaîne de caractères

```
with Read_Write;

procedure XX
is
x : String(1..50) := (others => ' ');
begin
....

Read_Write.Read(Header => "valeur de x" ,
                Value  => x);
....
end XX;
```

6.2.10 Sortie écran d'une chaîne de caractères

```
with Read_Write;

procedure XX
is
x : String(1..50) := (others => ' ');
begin
....

Read_Write.Write(Header => "valeur de x" ,
                 Value  => x);
....
end XX;
```

6.3 UTILISATION DE ADA.TEXT_IO

L'utilisation de ada.text_io pour la lecture est simple à quelques exceptions prés:

6.3.0.1 Lecture de nombre après instantiation:

6.3.0.1.1 Entier:

Si l'utilisateur entre 12 la valeur de I sera 12:

```
 Package integer_io is new ada.text_io.integer_io(integer);
....

 begin
 integer_IO.get (I);
```

Si on entre 12X3 pendant l'utilisation du code précédent, l'exception data_error sera «générée» et le caractère X ne sera pas effacé du tampon d'entrée. Toute nouvelle lecture d'une nombre est devenue impossible! il faut donc purger les caractères restants avant tout nouvel essai:

```
 Ada.text_io.Skip_Line;
```

6.3.0.1.2 Réel:

```
package My_Float_Io is new Ada.Text_Io.Float_Io(Float);
....
begin
My_float_IO.get(a_float);
```

6.3.0.1.3 Chaîne de caractères

Utilisation de ada_text_io.get_line

C'est la procédure de lecture d'une chaîne de caractères entrée par l'utilisateur, le code suivant peut paraître être à l'origine de DEUX entrées clavier successives, mais ……

Chaque get_line attend le «return» et renvoie: les caractères dans la chaîne et l'indice du dernier caractère dans la chaîne (ce qui n'est égal au nombre de caractères entrés qui si le tableau commence à 1!!!).

```
 String_1 : String (1..5);
 String_2 : String (1..5);
 Last_1   : Natural;
 Last_2   : Natural;

 Begin
.....
 String_1 := (others => ' ');-- ne jamais oublier
 String_2 := (others => ' ');-- ne jamais oublier

 ada.text_io.Get_Line (String_1, Last_1);

 ada.text_io.Get_Line (String_2, Last_2);
```

Tous les problèmes disparaissent si la longueur des chaînes est > au nombre de caractères entrés par l'utilisateur par ligne. C'EST CE QU'IL FAUT FAIRE sauf difficultés spéciales. Ici on utilise pour la démonstration des chaînes de 5 caractères

Si il y a moins de 5 caractères pour la première ligne par exemple ABC, string_1 contiendra «ABC »
NOTER bien les deux espaces à la fin!! et last_1 contiendra 3. le programme demandera ensuite la seconde ligne.

Si il y a 5 caractères pour la première ligne par exemple ABCDE, string_1 contiendra «ABCDE »
et last_1 contiendra 5. String_2 contiendra « » (5 espaces)
et last_2 contiendra 0 qui est le flag pour chaîne entrée vide (test facile pour «return» sans caractère avant).

Si il y a plus de 5 caractères pour la première ligne par exemple ABCDEFGH, string_1 contiendra «ABCDE » et last_1 contiendra 5. String_2 contiendra «FGH » NOTER bien les deux espaces a la fin!! et last_2 contiendra 3..

Pour purger entre les deux «get_line» utiliser la ligne de code suivante:

```
ada.text_io.Set_Col (ada.text_io.Current_Input, 1);
```

vous pouvez vérifier avec le programme w.adb

```
with Ada.Text_Io;

 procedure W
   is

   String_1 : String(1 .. 5);
   String_2 : String(1 .. 5);
   Last_1   : Natural;
   Last_2   : Natural;
   begin

   String_1 := (others => ' ');-- ne jamais oublier
   String_2 := (others => ' ');-- ne jamais oublier

   Ada.Text_Io.Put("string_1 ? ");

   Ada.Text_Io.Get_Line(String_1, Last_1);

   Ada.Text_Io.Put("string_2 ? ");

   Ada.Text_Io.Get_Line(String_2, Last_2);

   Ada.Text_Io.New_Line;

   Ada.Text_Io.Put_Line('>'& String_1 & "< last_1 " & Integer'Image(Last_1));

   Ada.Text_Io.Put_Line('>'& String_2 & "< last_2 " & Integer'Image(Last_2));

 end W;
```

qui donne les résultats suivants:

2 entrées de moins de 5 caractères le programme demande les deux chaînes

```
string_1 ? ABC
string_2 ? DE

>ABC  < last_1  3
>DE   < last_2  2
```

1 entrée de 5 caractères => **le programme demande une seule chaîne et se termine en sautant le second get_line:**

```
string_1 ? ABCDE
string_2 ?
>ABCDE< last_1  5
```

```
>     < last_2  0
```

1 entrée de plus de 5 caractères => le programme demande une seule chaîne et se termine en sautant le second get_line:

```
string_1 ? ABCDEFGH
string_2 ?
>ABCDE< last_1  5
>FGH  < last_2  3
```

Essayez pour vous même en commentant la ligne qui initialise les chaînes avec des caractères. Que se passe-t-il?

6.3.1 Sortie écran

6.3.1.1 Principes généraux

La sortie écran d'une variable peut être simple si aucun format précis n'est nécessaire, Dans ce cas on utilise l'attribut **image** qui transforme en chaîne de caractères facile à utiliser

6.3.1.2 Chaîne de caractères

```
Chaine :string (1..12);
.....
begin
....

       Ada.Text_Io.Put(Item => " La valeur est "
                       & chaine);
```

sur l'écran il apparaît:

La valeur est xyz

dans le cas ou vous voulez visualiser les espaces avant et après les autres caractères:

```
          Ada.Text_Io.Put(Item => " Le valeur est >"
                       & chaine
                        & '<');
```

sur l'écran il apparaît:

La valeur est > xyz   <
Ce qui indique la présence d'espaces.

Vous pouvez aussi utiliser Read_Write (annexe 1)

```
with Read_Write

procedure XX
is
x:Float;
begin
....
read_write.write(x);
read_write.write("valeur de x" , x);
....
end XX;
```

6.3.1.3 Entier

```
K : integer;
.....
begin
....

       Ada.Text_Io.Put(Item => " La valeur de k est "
                       & integer'image(k));
```

sur l'écran il apparaît:

La valeur de K est 3

Vous pouvez aussi utiliser Read_Write (annexe 1)

```
with Read_Write

procedure XX
is
x:Integer;
begin
....
read_Write.write(x);
read_write.write(rvaleur de x , x);
....
end XX;
```

6.3.1.4 Réels

6.3.1.4.1 Sans mise en forme

```
K : float;
.....
begin
....
k:=3.2;

       Ada.Text_Io.Put(Item => " La valeur de k est "
                       & float'image(k));
```

sur l'écran il apparaît:

La valeur de K est 3.20000E+00

6.3.1.4.2 Avec mise en forme

La procédure put pour les réels est définie en terme de générique (voir plus loin) de façon à ce que ce soit la même quelque soit le type de réel .

```
procedure Put(Item : in Num;
              Fore : in Field := Default_Fore;-- 2
              Aft  : in Field := Default_Aft; --  nombre de chiffres -1
              Exp  : in Field := Default_Exp); -- 3
```

Cette procédure écrit la valeur du paramètre item en utilisant le format de sortie défini par Fore, AFT et EXP.
Fore est le nombre de chiffes avant la virgule
Aft est le nombre de chiffres après la virgule
Exp est le nombre de chiffres de l'exposant

Si le réel est négatif, un signe – est inclus. Si EXP a pour valeur zéro, la partie entière du nombre contient autant de chiffres que nécessaire (la valeur de fore pourra être ignorée) ou contiendra le chiffre zéro si le nombre n'a pas de partie entière. Si EXP a une valeur plus grande que zéro, la partie entière écrite par cette procédure ne continent qu'un chiffre.
Dans ces deux cas, si la partie entière a moins de «FORE» chiffres des blancs seront ajoutés (pour aligner le point décimal sur plusieurs lignes successives).

Le nombre de chiffres des la partie décimale est donné par AFT (1 si aft=0) avec arrondi automatique à la précision demandée.

```
with Ada.Text_Io;
 procedure M_03
 is
 package My_Float_Io is new Ada.Text_Io.Float_Io(Float);
 X : Float := 123.123456;
 begin
 My_Float_Io.Put(Item => X,
                 Fore => 1,
                 Aft  => 2,
                 Exp  => 2);

 Ada.Text_Io.New_Line;

 My_Float_Io.Put(Item => X,
                 Fore => 1,
                 Aft  => 5,
                 Exp  => 3);

 Ada.Text_Io.New_Line;

 My_Float_Io.Put(Item => X,
                 Fore => 1,
```

```
                Aft  => 2,
                Exp  => 0);
 Ada.Text_Io.New_Line;
 end M_03;
```

Le résultat écran est:

```
F:\ada\current\w\miage>m_03
1.23E+2
1.23123E+02
123.12
```

6.3.1.5 Énumération / booléens

```
Type enum_type is (un,deux,trois);
K : enum_type;
B : boolean;
.....
begin
....

       Ada.Text_Io.Put(Item => " La valeur de k est "
                       & enum_type'image(k));
```

sur l'écran il apparaît:

La valeur de K est UN

```
Ada.Text_Io.Put(Item => " La valeur de b est "
                       & boolean'image(k));
```

sur l'écran il apparaît:

La valeur de K est «true».

6.4 LECTURE ET ÉCRITURE DE FICHIERS

6.4.1 Lecture de fichier texte

Voici un exemple de code

```
with Ada.Text_Io;
procedure Rw
   is
   Code_File : Ada.Text_Io.File_Type;
   Elem      : String(1 .. 100) := (others => ' ');
   Dummy     : Natural;
   begin
    -- ouverture du fichier à lire
   Ada.Text_Io.Open(File => Code_File,
                    Mode => Ada.Text_Io.In_File,
                    Name => "rw.adb");
  -- boucle sur toutes les lignes du fichier
   while not Ada.Text_Io.End_Of_File(Code_File) loop
    -- remplir avec des espaces
      Elem := (others => ' ');
    -- lire la ligne du fichier, dummy etant l'indice du dernier caractere lu
      Ada.Text_Io.Get_Line(File => Code_File,
                           Item => Elem,
                           Last => Dummy);
    --*********************************************************
    -- **** DEBUG ÉCRITURE DE CHAQUE LIGNE LUE SUR L'ECRAN ***
    --*********************************************************

      Ada.Text_Io.Put_Line(Elem(Elem'First .. Dummy));
    -- passer a la ligne suivante
   end loop;
  -- fermeture du fichier
   Ada.Text_Io.Close(File => Code_File);
end rw;
```

6.4.2 Écriture de fichier texte

Voici un exemple de code:

```
with Ada.Text_Io;
procedure Rw
   is
   Code_File : Ada.Text_Io.File_Type;

   begin

  -- création d'un fichier texte
   Ada.Text_Io.Create(File => Code_File,
                      Mode => Ada.Text_Io.Out_File,
                      Name => "test.txt");
  -- boucle pour écrire 3 lignes identiques
   for I in 1 .. 3 loop
    -- écriture d'une ligne de texte
      Ada.Text_Io.Put_Line(File => Code_File,
                           Item => " test test test");
   end loop;
  -- fermeture du fichier
   Ada.Text_Io.Close(File => Code_File);
end Rw;
```

7 LES ROUTINES: PROCEDURES ET FONCTIONS

En général, les programmes à écrire ne se limitent pas à quelque dizaines de lignes et doivent donc être divisés en plusieurs parties différentes. Se pose aussitôt le problème de la gestion des types de variable définis par l'utilisateur et commun à plusieurs routines. Ada a prévu ces problèmes et dispose des possibilités suivantes:
Regroupement de plusieurs fonctions et procédure dans un "package" lui même dédouble en deux unités:
La spécification (extension ads) contenant:

Les types de variable définis par l'utilisateur et commun à plusieurs routines **externes**.
La spécification des procédure et fonctions accessibles à des routines externes.

Le "body" (extension adb) contenant:

Le code des procédure et fonctions définies dans la spécification et donc accessible à des routines externes.
Le code des procédure et fonctions internes au "body" et donc inaccessible à l'extérieur.

7.1 PREMIER EXEMPLE: LA SOMME D'UN TABLEAU D'ENTIERS

Cet exemple calcule la somme du contenu d'un tableau d'entiers et affiche le résultat.
Le fichier a.ads est la spécification d'un package nommé "a" qui ne contient que la définition du type du tableau. Ce type de variable, défini par l'utilisateur, sera utilisé dans d'autre routines. Ce package "a" **NE POSSÈDE PAS DE "BODY"** il ne sert qu'a définir des variables communes!

Le fichier a.ads contient:

```
package A
is
Remarquer que cette définition ne précise pas la taille du tableau: cela laisse plus de facilité pour son usage
type My_Array_Type is array (Integer range <>) of Integer;
end A;
```

Le fichier c.ads est la spécification d'un package nommé "c" qui contient la définition de deux procédure : l'une calcule la somme et l'autre affiche le résultat. Leur fonctionnement interne sera implémenté dans le "body" correspondant et sera inaccessible pour les utilisateurs de ce "package". Noter son utilisation du package "a" pour la définition du type du tableau.

Le fichier c.ads contient:

```
with A;
package C
is
procedure Sum(Table : in     A.My_Array_Type;
              Into  :    out Integer);

procedure Print_Result(R : in     Integer);
end C;
```

Le fichier c.adb est le "body" d'un package nommé "c" .Il contient le code des deux procédures définies dans la spécification correspondante (c.ads): l'une calcule la somme et l'autre affiche le résultat.

Le fichier c.adb contient:

```
with Ada.Text_Io;
package body C
is
procedure Sum(Table : in     A.My_Array_Type;
              Into  :    out Integer)
   is
   begin

   Into := Table(Table'First);

   for I in 1 + Table'First .. Table'Last loop

      Into := Into + Table(I);
   end loop;

end Sum;

procedure Print_Result(R : in     Integer)
   is
   begin
   Ada.Text_Io.Put("result" & Integer'Image(R));
end Print_Result;
end C;
```

Le fichier main.adb contient le programme principal. Noter sa simplicité! Appel au calcul puis appel à l'affichage.

Le fichier main.adb contient:

```
with A;
with C;

procedure Main
  is
  Result : Integer;
Déclaration d'un tableau t de type my_array_type défini dans le package "a" et initialisé à 1 et 2
  T:  A.My_Array_Type(1..2):=(1,2);
  begin
Appel à la procédure sum dans le package "C "avec comme arguments un tableau contenant 1 et 2, le résultat étant mis dans result.
  C.Sum(Table => (1, 2),
        Into  => Result);
Appel à la procédure Print_Result dans le package "C" avec comme argument result
  C.Print_Result(R => Result);
Appel à la procédure sum dans le package "C" avec comme arguments le tableau T, le résultat étant mis dans result.
  C.Sum(Table => T,
        Into  => Result);
Appel à la procédure Print_Result dans le package "C" avec comme argument result
  C.Print_Result(R => Result);
end Main;
```

7.2 SECOND EXEMPLE: MANIPULATION DES BITS, APPLICATION À LA LECTURE D'UN FICHIER TEXTE PAR BLOC POUR CODAGE

Bien que d'un emploi peu fréquent, il est possible de manipuler les bits individuellement. Il faut simplement pour cela déclarer un type modulo:

```
Type mod_32 is mod 2**32;
```

Ce type servira dans tout ce qui suit .

7.2.1 Utilisation des opérateurs logiques

Les opérateurs logique classiques : ou et non ou exclusif fonctionnent pour chaque bit individuellement:

```
A: Mod_32;
B: Mod_32;
C: Mod_32;
....
```

L'écriture des values en binaire et hexadécimal (ou toute autre base) est simple: base#valeur#

```
C := 16#F# -- 15
A := 2#101#;-- 5
```

A contient les 32 bits suivant: 00000000000000000000000000000101 noter l'usage de ne pas écrire les zéros non significatifs à gauche.

```
B:=2#010#;--2

C:=A and B;
```

C contient 0 car l'opérateur est appliqué bit après bit.

A 1 0 1

B 0 1 0
C 0 0 0
Pour les bits bleus 1 (A) and 0 (B) donne 0
Pour les bits rouge 0 (A) and 1 (B) donne 0
Pour les bits bleus 1 (A) and 0 (B) donne 0

Le même mécanisme fonctionne avec les autres opérateurs.

7.2.1.1 Masque

Il est souvent nécessaire d'isoler, d'effacer ou de tester un ou plusieurs bits dans un mot. Cela se fait avec un masque;

Soit le masque:
```
M : constant Mod_32:=2#1#;
C:=A and M;
```
Tous les bits de A sont maintenant à zéro sauf le bit tout à fait à droite qui reste inchangé.

7.2.2 Assignation

Pour mettre un ou plusieurs bits à une valeur donnée, il est préférable de faire en deux temps:
Mise a zéro avec un masque et l'opérateur et
Assignations de ces bits avec l'opérateur ou.

Exemple pour mettre les bits de rang 3 et 4 de A aux valeurs des bits de rang 3 et 4 de B
```
M_1 : constant Mod_32:=2#11111111111111111111111111110011#;
M_2 : constant Mod_32:=2#00000000000000000000000000001100#;
A:= A and M_1;
C:= B and M_2
A:=A or C;
```

Exemple: tester si un au moins des 4 bits de rang 28 à 31 est à 1
```
if A and 16#F000#  /=0 then
....
```

7.2.3 Décalage gauche et droite

Les travaux au niveau des bits impliquent souvent de décaler avec un remplissage de zéros: L'utilisation de multiplication et de division par des puissances de 2 permet d'effectuer ces opérations.

```
B:=2#010#;--2
B:= B*2**3;
```
B contient maintenant les bits 10000

```
B:=2#01111#;--2
B:= B / 2**3;
```
B contient maintenant le bit 1

7.2.3.1 Rotation

Le but est d'effectuer un permutation circulaire des 32 bits d'un mot, La direction ainsi que le nombre de permutations est un paramètre d'entrée de la fonction:

```
Function Rotate_Right (X     : in Mod_32;
                       Steps : in positive)
return Mod_32
is

Left  : Mod_32;
Right : Mod_32;

begin

Right := X /2** Steps;
Left  := X / 2 ** (32 - Steps);
Return left or right;
end Rotate_Right ;



Function Rotate_Left (X     : in Mod_32;
                      Steps : in positive)
return Mod_32
is

Left  : Mod_32;
Right : Mod_32;
```

```
begin

Left  := X * 2** Steps;
Right := X / 2 **(32 - Steps);
Return left or right;
end Rotate_Left ;
```

7.2.4 Détails du stockage des entiers, caractères et réels dans un mot de 32 bits

Tout ce qui est discuté précédemment ne préjuge pas de la réalité physique représentée dans l'ordinateur. Cette réalité apparaît dans les conditions discutées ci dessous.

7.2.4.1 Partage de zone de mémoire

L'un des outils mis à disposition par le langage permet de l'affectation bit pour bit entre deux types de variables si le nombre de bits utilisé pour les représenter est identique.
Par exemple:

```
type byte is mod 2**8;
for Byte'size use 8;
type character_table is array (0..3) of character;
type byte_table is array (0..3) of bytes;
```

Les types mod_32, character_table et byte_table peuvent utiliser l'affectation bit par bit car leur longueur est de 32 bits. Cette conversion se fait en traduisant d'un type dans l'autre bit pour bit. Pour cela il faut définir une fonction qui permet cette conversion en indiquant le type de départ et le type d'arrivée.
Voici un exemple:

```
with Ada.Text_Io;
with Unchecked_Conversion;
procedure X
  is
  type Byte is mod 2 ** 8;
  for Byte'Size use 8;
  type Character_Table is array (0 .. 3) of Character;

  type Byte_Table is array (0 .. 3) of Byte;
  type Mod_32 is mod 2 ** 32;

 -- conversion entre   byte_array et mod_32

  function Byte_Table_To_Mod_32 is new Unchecked_Conversion(Source => Byte_Table,
                                                            Target => Mod_32);

  function Mod_32_To_Byte_Table is new Unchecked_Conversion(Source => Mod_32,
                                                            Target => Byte_Table);
 -- Conversion  entre character_table   et mod_32

  function Character_Table_To_Mod_32 is new Unchecked_Conversion(Source => Character_Table,
                                                                 Target => Mod_32);

  function Mod_32_To_Character_Table is new Unchecked_Conversion(Source => Mod_32,
                                                                 Target => Character_Table);
```

Routines de sortie écran

```
  package Mod_32_Io is new Ada.Text_Io.Modular_Io(Mod_32);
  package Byte_Io is new Ada.Text_Io.Modular_Io(Byte);
```

Mettons une valeur binaire dans x

```
  X           : Mod_32 := 2#01100100011000110110001001100001#;
  Character_T : Character_Table;
  Byte_T      : Byte_Table;
  begin
  --|
  --| to screen   base#xxxxxxxx#
  --|
  Mod_32_Io.Put(Item => X,
                Base => 2);
  Ada.Text_Io.New_Line;
```

La sortie écran est : 2#01100100 01100011 01100010 01100001# (le 0 de gauche et les espaces sont ajoutés pour la lisibilité)

Traduction bit pour bit dans le tableau de 0 à 3 de caractères

```
  Character_T := Mod_32_To_Character_Table(X);

  Ada.Text_Io.Put_Line(Character_T(0) & Character_T(1) & Character_T(2) & Character_T(3));
```

La sortie écran est : abcd

Assignation des mêmes caractères

```
  Character_T(0) := 'A';
  Character_T(1) := 'B';
```

```
    Character_T(2) := 'C';
    Character_T(3) := 'D';


    X := Character_Table_To_Mod_32(Character_T);

    --|
    --| to screen   base#xxxxxxxx#
    --|
    Mod_32_Io.Put(Item => X,
                  Base => 2);
    Ada.Text_Io.New_Line;
```

La sortie écran est : 2#01000100 01000011 01000010 01000001# (le 0 de gauche et les espaces sont ajoutés pour la lisibilité)
CELA CONFIRME QUE LE CARACTÈRE A BIEN QUE LE PREMIER DU TABLEAU EST STOCKÉ A DROITE DANS LE MOT DE 32 BITS (BITS 0 À 7)

```
    --|
    Byte_T := Mod_32_To_Byte_Table(X);
    for I in Byte_T'Range loop
       Byte_Io.Put(Item => Byte_T(I),
                   Base => 2);
    end loop;
```

La sortie écran est : 2#1000001#2#1000010#2#1000011#2#1000100#
CELA CONFIRME QUE LE byte (0) BIEN QUE LE PREMIER DU TABLEAU EST STOCKÉ A DROITE DANS LE MOT DE 32 BITS (BITS 0 À 7)

EFFAÇONS les 8 bits de 0 à 7 à D R O I T E D U N O M B R E pour voir l'effet sur les bytes:

```
      X:=X and 16#FFFFFF00#;
    --|
    --| to screen   base#xxxxxxxx#
    --|
    Mod_32_Io.Put(Item => X,
                  Base => 2);
    Ada.Text_Io.New_Line;
```

La sortie écran est : 2#00000000 01000011 01000010 01000001# (les 0 de gauche et les espaces sont ajoutés pour la lisibilité)
CELA CONFIRME QUE LE CARACTÈRE 'A' BIEN QUE LE PREMIER DU TABLEAU EST STOCKÉ A DROITE DANS LE MOT DE 32 BITS (BITS 0 À 7)

```
    --|
    Byte_T := Mod_32_To_Byte_Table(X);
    for I in Byte_T'Range loop
       Byte_Io.Put(Item => Byte_T(I),
                   Base => 2);
    end loop;
```

La sortie écran est : 2#0# 2#1000010# 2#1000011# 2#1000100#
CELA CONFIRME QUE LE byte (0) BIEN QUE LE PREMIER DU TABLEAU EST STOCKÉ A DROITE DANS LE MOT DE 32 BITS (BITS 0 À 7)

```
  end X;
```

L'explication de ce comportement étrange est indiquée ci dessous:

7.2.4.2 Transformation little endian big endian

La façon réelle utilisée pour stocker les bits dans la mémoire physique:

Selon la marque du processeur de l'ordinateur utilisé, la façon dont les données sont stockées dans la mémoire peut se faire de deux méthodes différentes appelées "little endian" et big "endian". **Cette différence est invisible tant que les données ne sont pas échangées en binaire à travers un fichier** ou que l'on ne s'intéresse pas aux détails de la position des bits respectifs dans la mémoire. Cette différence apparaît au niveau du découpage par bytes et est valable que le stockage concerne un entier, un réel ou une chaîne de caractères. Prenons l'exemple du stockage de caractères dans la mémoire d'un ordinateur utilisant la méthode "little endian". Chaque caractère est codé sur un byte (8 bits) en utilisant le code international (ASCII). Ce code, défini il y a de nombreuses années, est maintenant très communément utilisé.
Supposons un emplacement mémoire de 32 bits (soit 4 caractères) et par exemple la chaîne "abcd". Numérotons les 32 bits de 0 à 31 tel que le nombre représenté par ces 32 bits soit un entier sans signe défini par:

$$\text{Nombre} = Bit\_0 * 2^0 + bit\_1*2^1 + bit\_3*2^2 + ... + bit\_i * 2^i + ... + bit\_31*2^{31}$$

Exemple avec une machine "little endian" (Intel par exemple)
Pour mieux comprendre observons en binaire sur 32 bits.

Bit 31 bit 0
abcd en binaire sur 32 bits :01100100011000110110001001100001

La lettre a code ASCII 01100001 (hexadecimal 31) est **stockée dans le byte de droite** dans les bits numérotés 7 à 0 et ainsi de suite.

Le même exemple avec une machine "big endian" (Motorola par exemple)
Pour mieux comprendre observons en binaire sur 32 bits.

Bit 31 bit 0
abcd en binaire sur 32 bits :01100001 01100010 01100011 01100100

La lettre a code ASCII 01100001 (hexadecimal 31) est **stockée dans le byte de gauche** dans les bits numérotés 31 à 24 et ainsi de suite.

Voici deux version du code de la conversion de l'un dans l'autre et réciproquement:

```
with Ada.Text_Io;
with Unchecked_Conversion;
........
type Mod_32 is mod 2 ** 32;
```

La première version utilise le décalage, les masques et les opérateurs logiques

```
function Little_Big_Endian(X : in     Mod_32)
   return Mod_32
   is
   Mod_32_0 : Mod_32;
   Mod_32_1 : Mod_32;
   Mod_32_2 : Mod_32;
   Mod_32_3 : Mod_32;
   begin

   Mod_32_0 := (X and 16#FF000000#) / 2 ** 24;
   Mod_32_1 := (X and 16#00FF0000#) / 2 ** 16;
   Mod_32_2 := (X and 16#0000FF00#) / 2 ** 8;
   Mod_32_3 := X and 16#000000FF#;

   return
      Mod_32_3 * 2 ** 24
   or Mod_32_2 * 2 ** 16
   or Mod_32_1 * 2 ** 8
   or Mod_32_0;

end Little_Big_Endian;
```

La seconde version utilise le partage mémoire

```
function Little_Big_Endian_1(X : in     Mod_32)
   return Mod_32
   is
   type Byte is mod 2 ** 8;
   for Byte'Size use 8;
   type Byte_Table is array (0 .. 3) of Byte;
   function Byte_Table_To_Mod_32 is new Unchecked_Conversion(Source => Byte_Table,
                                                             Target => Mod_32);

   function Mod_32_To_Byte_Table is new Unchecked_Conversion(Source => Mod_32,
                                                             Target => Byte_Table);
   Byte_T : Byte_Table;
   Result : Byte_Table;

   begin

   Byte_T := Mod_32_To_Byte_Table(X);

   for I in Byte_T'Range loop
      Result(Byte_T'Last - I) := Byte_T(I);
   end loop;

   return Byte_Table_To_Mod_32(Result);

end Little_Big_Endian_1;
```

7.2.5 La lecture d'un fichier texte par bloc pour codage

La plupart des systèmes de codage (AES, LZW,..) ou de calcul de hashing (SHA, MD5, ...) supposent une lecture d'un fichier texte façon big endian. En ada cela suppose une lecture de fichier et/ou un changement little endian big endian. Dans ce cas, il faut pour que le code soit lisible et modifiable utiliser un itérateur pour la lecture. Le fichier à lire sera considéré comme un objet muni d'opérations:

- Init_reading(file_name): Tente l'ouverture du fichier en lecture et initialise l'itérateur.
- Assign_To_Current_State(state): Charge les données courantes dans « state ».
- Reading_Over : Renvoi un booléen qui est vrai si il n'y a plus de données à lire.

- Get_Next_State: Avance d'un « block » dans le fichier.

L'utilisation des opérations ci-dessus peut se résumer comme suit:

```
Init_reading(file_name)
while not Reading_Over loop
Assign_To_Current_State(state)
TRAITEMENT DE « STATE »
.....
Get_Next_State
end loop;
```

De plus, le fichier pouvant être un fichier binaire ou un fichier texte, nous devons pour simplifier utiliser le package stream _io.
La question la plus intéressante à résoudre est comment traiter les fichiers dont la taille est un multiple de 16 bytes ou non.

Pour la partie écriture, tout sera considéré comme des bytes. Cette partie dispose des opérations suivantes:

- Init_Writing(File_Name): Tente l'ouverture du fichier en écriture
- Write(State) : Écrit par bytes

L'utilisation des deux s'écrit:

```
Init_reading(file_name)
Init_Writing(File_Name)
while not Reading_Over loop
Assign_To_Current_State(state)
TRAITEMENT DE « STATE »
.....
Write(State)
Get_Next_State
end loop
Close_File
```

Le code complet ci-dessous sera étudié attentivement pour découvrir comment utiliser le package stream _io:

```
package Rijndael_Data
  is
.......
  type Byte is mod 2 ** 8;
  for Byte'Size use 8;
  type State_Array_Type is array (Byte range <>, Byte range <>) of Byte;
  ...............
end Rijndael_Data;
```

```
with Ada.Streams.Stream_Io;
with Rijndael_Data;

package Rijndael_Cipher_Io
  is

  function Init_Reading(File_Name : in     String) return Boolean;

  procedure Assign_To_Current_State(State : in out Rijndael_Data.State_Array_Type);

  function Reading_Over return Boolean;

  procedure Get_Next_State;

  function Init_Writing(File_Name : in     String) return Boolean;

  procedure Write(State : Rijndael_Data.State_Array_Type);

  procedure Close_File;

  File_Not_Found : exception;

  private

  type Table_Of_Bytes_Type is array (0 .. 15) of Rijndael_Data.Byte;
  Current_Table_Of_Bytes  : Table_Of_Bytes_Type;
  Next_Table_Of_Bytes     : Table_Of_Bytes_Type;
  The_File_In             : Ada.Streams.Stream_Io.File_Type;
  The_File_Out            : Ada.Streams.Stream_Io.File_Type;
  File_Stream_In          : Ada.Streams.Stream_Io.Stream_Access;
```

```
   File_Stream_Out         : Ada.Streams.Stream_Io.Stream_Access;
   Done                    : Boolean := False;
   First_Time              : Boolean := True;
   Last_Block_In_File_Read : Boolean := False;

end Rijndael_Cipher_Io;




with Ada.Io_Exceptions;
with Rijndael_Data;

package body Rijndael_Cipher_Io
  is

  procedure Get_Next_State
     is
     --|
     --| the input file will be read by 16 bytes at a time because this is the state size
     --| if the file size is not a multiple of 16 the remaining bytes are set to ASCII.space
     --| this is not documented in the AES standard
     --| an exception handler is used to catch the reading of the last incomple set of 16 bytes
     --|
     Index : Natural := Current_Table_Of_Bytes'First;
     begin
     --|
     --| is it over ??     (written this way "done" is only true the second time)
     --|
     Done := Done or Last_Block_In_File_Read;
     --|
     --| clear table
     --|
     Current_Table_Of_Bytes := (others => 32);-- ascii.space is dec 32 (hex 20)
     Next_Table_Of_Bytes    := (others => 32);
     --|
     --| the first time in the program attemps to load two set of states
     --|  this way for the following read operations, it can cope with either
     --| the file size in bits modulo 128 is zero or not
     --|
     if First_Time
        then
        --|
        --| load 16 bytes into current_Table_Of_Bytes
        --|
        while Index in Current_Table_Of_Bytes'Range loop
           Rijndael_Data.Byte'Read(File_Stream_In,
                                   Current_Table_Of_Bytes(Index));
           Index := 1 + Index;
        end loop;
        First_Time := False;
        else
        Current_Table_Of_Bytes := Next_Table_Of_Bytes;
     end if;
     --|
     --| load 16 bytes into next_Table_Of_Bytes
     --|
     Index := Next_Table_Of_Bytes'First;
     while Index in Next_Table_Of_Bytes'Range loop
        Rijndael_Data.Byte'Read(File_Stream_In,
                                Next_Table_Of_Bytes(Index));
        Index := 1 + Index;
     end loop;
     --|
     --| well, end of file reached: reading over before
     --| the end of the reading of 16 bytes
     --|
     exception
     when Ada.Io_Exceptions.End_Error =>
     Last_Block_In_File_Read := True;
  end Get_Next_State;

  function Init_Reading(File_Name : in     String)
     return Boolean
     is
     begin
     --|
     --| close if previous file already used
     --|
     if Ada.Streams.Stream_Io.Is_Open(File => The_File_In)
        then
        Ada.Streams.Stream_Io.Close(File => The_File_In);
     end if;
     --|
     --| set to   false
     --|
     Done                    := False;
     Last_Block_In_File_Read := False;
     First_Time              := True;
     --|
     --| try opening the file
     --|
     Ada.Streams.Stream_Io.Open(File => The_File_In,
                                Name => File_Name,
                                Mode => Ada.Streams.Stream_Io.In_File);
     --|
```

```
      --| get stream access
      --|
      File_Stream_In := Ada.Streams.Stream_Io.Stream(File => The_File_In);
      --|
      --| fill first bytes from file into Table_Of_Bytes, ready for state
      --|
      Get_Next_State;
      --|
      --| file opened
      --|
      return True;
      --|
      --|   no such file ??
      --|
      exception
      when others =>
      Done := True;
      Ada.Streams.Stream_Io.Close(File => The_File_In);
      return False;
   end Init_Reading;

   procedure Assign_To_Current_State(State : in out Rijndael_Data.State_Array_Type)
      is
      use type Rijndael_Data.Byte;
      Index : Natural := Current_Table_Of_Bytes'First;
      begin
      --|
      --|  load from Table_Of_Bytes into state  by exchanging lines and columns
      --|
      for Line in State'Range(1) loop
         for Column in State'Range(2) loop
            State(Column, Line) := Current_Table_Of_Bytes(Index);
            Index               := 1 + Index;
         end loop;
      end loop;
   end Assign_To_Current_State;

   function Reading_Over
      return Boolean
      is
      begin
      return Done;
   end Reading_Over;
   --
   --
   --*****************************************************************
   --******************  W R I T I N G ***********************************
   --*****************************************************************

   function Init_Writing(File_Name : in     String)
      return Boolean
      is
      begin
      --|
      --| close if previous file already used
      --|
      if Ada.Streams.Stream_Io.Is_Open(File => The_File_Out)
         then
         Ada.Streams.Stream_Io.Close(File => The_File_Out);
      end if;
      --|
      --| try opening the file
      --|
      Ada.Streams.Stream_Io.Create(File => The_File_Out,
                                   Name => File_Name,
                                   Mode => Ada.Streams.Stream_Io.Out_File);
      --|
      --| get stream access
      --|
      File_Stream_Out := Ada.Streams.Stream_Io.Stream(File => The_File_Out);
      --|
      --| file created
      --|
      return True;
      --|
      --|   creation problem ??
      --|
      exception
      when others =>
      return False;
   end Init_Writing;

   procedure Write(State : Rijndael_Data.State_Array_Type)
      is
      use type Rijndael_Data.Byte;
      Dummy_State : Rijndael_Data.State_Array_Type := State;
      begin
      --|
      --|  exchange lines and columns
      --|
      for Line in State'Range(1) loop
         for Column in State'Range(2) loop
            Dummy_State(Line, Column) := State(Column, Line);
         end loop;
      end loop;
      --|
      --|  on disk
```

```
      --|
      Rijndael_Data.State_Array_Type'Write(File_Stream_Out,
                                           Dummy_State);
   end Write;

   procedure Close_File
      is
      begin
      --|
      --| close files
      --|
      Ada.Streams.Stream_Io.Close(File => The_File_Out);
      Ada.Streams.Stream_Io.Close(File => The_File_In);
   end Close_File;

end Rijndael_Cipher_Io;
```

7.3 PROGRAMMATION D'UN CODE À EXÉCUTER AU DÉBUT ET LA FIN DU DOMAINE DE VALIDITÉ D'UNE VARIABLE

Il peut arriver de vouloir exécuter un code au début du domaine de validité d'une variable (sa déclaration) et a la fin de cette période. Une librairie est prévue à cet effet: Ada.Finalization. Vous trouverez ci-dessous un code simple illustrant cette possibilité:

```
with Ada.Finalization;
package C
   is
   type My_Type is new Ada.Finalization.Controlled with
      record
      I : Integer;
   end record;
   procedure Initialize(Object : in out My_Type);
   procedure Finalize(Object   : in out My_Type);
end C;

with Ada.Text_Io;
package body C
   is
   procedure Initialize(Object : in out My_Type)
      is
      begin
      Ada.Text_Io.Put_Line("Initialize " & Integer'Image(Object.I));
   end Initialize;

   procedure Adjust(Object : in out My_Type)
      is
      begin
      Ada.Text_Io.Put_Line("Adjust " & Integer'Image(Object.I));
   end Adjust;

   procedure Finalize(Object : in out My_Type)
      is
      begin
      Ada.Text_Io.Put_Line("Finalize " & Integer'Image(Object.I));
   end Finalize;
end C;

with C;
with Ada.Finalization;
procedure Main
   is
début du domaine de validité de variable
   Variable : C.My_Type;
   begin
   Variable   := C.My_Type'(Ada.Finalization.Controlled with I => 0);
   Variable.I := Variable.I + 1;
fin du domaine de validité de variable
end Main;
```

8 GÉNÉRIQUE

8.1 UN CAS D'ÉCOLE

Prenons un exemple ridiculement simple: L'égalité de deux variables. Ce problème semble impliquer plusieurs procédures différentes, une pour chaque type de variable à comparer. Voici un code possible pour un record avec deux comparaisons différentes:

```
.........
    type My_Record
       is record
       A : Integer        := Integer'First;
       B : String(1 .. 5) := (others => ' ');
    end record;
```

```
function Egal_1(X : in     My_Record;
                Y : in     My_Record)
   return Boolean
   is
   begin
   return X.A = Y.A;
end Egal_1;
```

```
function Egal_2(X : in     My_Record;
                Y : in     My_Record)
   return Boolean
   is
   begin
   return X.B = Y.B;
end Egal_2;
```

La différence réside seulement dans la comparaison! De plus ce code est très court, que faire si le code est beaucoup plus long . Comment ne pas dupliquer ce code. C'est là le GROS défaut du code précédent: Il ne peut fonctionner que si les données sont des records du type défini et la comparaison implique une duplication. Que faire si je veux le même programme pour des entiers, des réels, des énumérations, des caractères, des records, des types non encore connus. La solution du langage Ada est le générique.

8.1.1 La solution (ici elle est plus longue que la duplication mais c'est un exemple simpliste!) se compose de trois fichiers:

8.1.1.1 Une partie générique, fichier s.ads

```
generic
Ici on indique que le type de données est totalement inconnu et sera au choix de l'utilisateur
type Record_X is private;
L'instruction suivante précise que la fonction = SERA écrite par l'utilisateur
with function "=" (X : in Record_X;
                  Y : in     Record_X) return Boolean;
Déclaration de la fonction générique
function S(X         : in     Record_X;
           Y         : in     Record_X) return Boolean;
```

8.1.1.2 Une partie générique, fichier s.adb

```
La fonction de comparaison
function S(X : in     Record_X;
           Y : in     Record_X)
  return Boolean
  is
  begin
  return X = Y;
end S;
```

8.1.1.3 Un exemple de procédure d'utilisation, fichier d.adb

```
with S;
procedure D
  is
Le type de données est un "record" contenant un entier et une chaîne de caractères
  type My_Record
     is record
     A : Integer        := Integer'First;
     B : String(1 .. 5) := (others => ' ');
  end record;
L'utilisateur DOIT définir la fonction =
Cette fonction défini = en comparant les deux entiers
  function Egal_1(X : in     My_Record;
                  Y : in     My_Record)
     return Boolean
     is
     begin
     return X.A = Y.A;
```

```
  end Egal_1;
```
L'utilisateur DOIT définir la fonction =
Cette fonction défini = en comparant les deux chaînes de caractères
```
  function Egal_2(X : in     My_Record;
                  Y : in     My_Record)
     return Boolean
     is
     begin
     return X.B = Y.B;
  end Egal_2;
```
Il faut maintenant associer (une instantiation) la fonction générique avec sont type de données et la fonction contenant la définition de l'opérateur =

Première instantiation en utilisant la comparaison des entiers car la fonction Egal_1 est le paramètre correspondant :

```
  function Egal_A is new S(Record_X => My_Record,
                           "="      => Egal_1);
```

Seconde instantiation en utilisant la comparaison des chaînes de caractères car la fonction Egal_2 est le paramètre correspondant :

```
  function Egal_B is new S(Record_X => My_Record,
                           "="      => Egal_2);
```

Deux fonctions différentes, Egal_A et Egal_B sont à notre disposition pour effectuer la comparaison de notre choix.

Déclaration et initialisation des variables

```
   My_Rec_1 : My_Record := (A        => 1,
                            B        => "12345");

  My_Rec_2 : My_Record := (A        => 2,
                           B        => "12345");

  X : Boolean;
  Y : Boolean;

  begin

  X := Egal_A(X => My_Rec_1,
              Y => My_Rec_2);

  Y := Egal_B(X => My_Rec_1,
              Y => My_Rec_2);
end D;
```
Après exécution du code, x et y sont respectivement faux et vrai.

8.2 L'INTÉRÊT DU GÉNÉRIQUE

Le générique
- **La même logique**
- **Avec les mêmes opérations**
- **Mais sans que l'objet soit défini d'avance**.

8.3 LES VARIABLES GÉNÉRIQUES

Les solutions suivantes sont disponibles pour les variables de type (plus ou moins) inconnu.

8.3.1 Les variables de type private

Ces variables ont un type complètement inconnu et par conséquent aucun opérateur n'est défini.

```
type Object_Type is private;
```

8.3.2 Les variables de type discrètes

Ces variables peuvent être soit des entiers, une énumération, un booléen ou des caractères. Ce type est souvent utilisé comme indice d'un tableau. Les opérateurs usuels des variables de type discret sont disponibles, en particulier, l'opérateur (+1) passage à l'élément suivant et l'opérateur (-1) passage à l'élément précédent sont définis. Ils sont utilisés, en particulier, pour la gestion des indices génériques dans les boucles:
x:= Index_Type'succ(x);-- pour +1
x:= index_type'pred(x) ;-- pour -1

```
 type  is (<>);
```

Un type de tableau de contenu inconnu et d'indice entier, énumération, caractères pourrait être codé:

```
type Object_Array_Type is array (Index_Type range <>) of Object_Type;
```

8.3.3 Les variables de type entier

Un entier (normal ou long) sans connaissance des détails (début et fin)

```
type entier_Type is range <>;
```

8.3.4 Les variables de type Modulo

Un modulo sans connaissance des détails (début et fin)

```
type entier_Type is mod <>;
```

8.3.5 Les variables de type Réel

Ces types correspondent exactement aux possibilités du langage.

```
type Coefficient is digits 10 range -1.0 .. 1.0;
```
deviendra
```
type Coefficient is digits <>;
```

```
type Volt is delta 0.125;
```
deviendra
```
type Volt is delta <>;
```

```
type Money is delta 0.01 digits 15;
```
deviendra
```
type Money is delta <> digits <>;
```

8.4 LES PROCÉDURES ET FONCTIONS GÉNÉRIQUES

8.4.1 Les changements dans le code

Certaines modifications sont nécessaire pour prendre en compte:

- Le cas des indices de tableaux car les opérations arithmétiques (+ - * /) ne sont pas définies directement. Il faut utiliser les **attributs** pour transformer les indices en «rang» dans le tableau, faire l'arithmétique sur le «rang» dans le tableau puis transformer en indice.
- Les opérations logique car les opérateurs >, < <= et >=, …. peuvent ne pas être définis. Le générique comportera des fonctions / procédures, incomplètes pour ces opérations. L'utilisateur, au moment de l'emploi du générique, précisera ces procédures.

8.4.2 La spécification

Le code commence par «generic» puis a la même architecture que les spécifications habituelles;

Les types sont déclarés comme décrit plus haut. Les opérateurs logiques n'existent plus pour "private", il faut créer le code qui demandera à l'utilisateur ce créer une fonction à cet effet:

- Une pour l'opérateur >
- Une pour l'opérateur <

```
with function ">" (X : in Object_Type;
                   Y : in     Object_Type) return Boolean is <>;

 with function "<" (X : in Object_Type;
                   Y : in     Object_Type) return Boolean is <>;
```

Souvenez vous que le contraire de l'opérateur > n'est pas l'opérateur < mais <= .

Par exemple pour trouver:

- Le plus petit élément d'un tableau et son indice associé
- Le plus grand élément d'un tableau et son indice associé
- L'élément situé au milieu d'un tableau
- Le nombre d'élément dans ce tableau

on peut écrire pour la spécification fichier demo.ads:

```
generic
 type Object_Type is private;
 type Index_Type is (<>);
 type Object_Array_Type is array (Index_Type range <>) of Object_Type;

 with function ">" (X : in Object_Type;
                    Y : in     Object_Type) return Boolean is <>;

 with function "<" (X : in Object_Type;
                    Y : in     Object_Type) return Boolean is <>;

 package demo
 is

 procedure   bornes(Tableau         : in     Object_Array_Type;
                    Min             :    out Object_Type;
                    Max             :    out Object_Type;
                    Value_At_Middle :    out Object_Type;
                    Indice_Min      :    out Index_Type;
                    Indice_Max      :    out Index_Type;
                    Size            :    out Integer);
 end Demo;
```

8.4.3 Le body

Les éléments non présents dans les paramètres d'appel sont déclarés:

```
     Indice_At_Middle : Index_Type;
```

Pour la valeur minimum il faut utiliser l'attribut first pour avoir l'indice du premier élément, de même pour le dernier. C'est l'initialisation des valeurs .

```
     Indice_Min       := Tableau'First;
     Min              := Tableau(Indice_Min);
     Indice_Max       := Tableau'last;
     Max              := Tableau(Indice_Max);
```

L'attribut «pos» renvoie l'offset en entier correspondant à l'indice 0 pour le premier, 0 pour le second,……

Exemple:
Un tableau d'indice entier de –3 .. +3 dans ce cas index_type'pos(0) renvoie l'entier 3
Type z is(ab,bc,de,fg) Un tableau d'indice de type z dans ce cas cas z'pos(de) renvoie l'entier 2
L'attribut «val» fait l'opération inverse, il revoie l'indice du tableau à partir du «rang» de l'indice

```
     Size             := Index_Type'Pos(Indice_Max) - Index_Type'Pos(Indice_Min) + 1;
     Indice_At_Middle := Index_Type'Val(Size / 2);
     Value_At_Middle  := Tableau(Indice_At_Middle);
```

La recherche du max et du min est classique. Remarquez la syntaxe de la boucle

```
for Index in Tableau'Range loop
     ...
end loop;
```

Le code est dans le fichier demo.adb

```
package body Demo
   is
   procedure Bornes(Tableau         : in     Object_Array_Type;
                    Min             :    out Object_Type;
                    Max             :    out Object_Type;
                    Value_At_Middle :    out Object_Type;
                    Indice_Min      :    out Index_Type;
                    Indice_Max      :    out Index_Type;
                    Size            :    out Integer)
     is
     Indice_At_Middle : Index_Type;
     begin
     Indice_Min       := Tableau'First;
     Min              := Tableau(Indice_Min);

     Indice_Max       := Tableau'Last;
     Max              := Tableau(Indice_Max);

     Size             := Index_Type'Pos(Indice_Max) - Index_Type'Pos(Indice_Min) + 1;
     Indice_At_Middle := Index_Type'Val(Size / 2);
     Value_At_Middle  := Tableau(Indice_At_Middle);

     for Index in Tableau'Range loop
       if Tableau(Index) < Min
         then
         Indice_Min := Index;
         Min        := Tableau(Index);
       end if;
       if Tableau(Index) > Max
         then
```

```
        Indice_Max := Index;
        Max        := Tableau(Index);
      end if;
    end loop;
  end Bornes;
end Demo;
```

8.5 LE PROGRAMME PRINCIPAL

Il utilise le générique précèdent et imprime les résultats pour deux données différentes. Détaillons ensemble sa syntaxe et sa structure. Il est divisé en deux types de données différentes ***«utilisant le même générique».***

8.5.1 Premier type de données

L'index du tableau est un entier de -2 à 3
`subtype U_Index_Type is Integer range - 2 .. 3;`
Le contenu du tableau est un entier de 1 à 5
`subtype U_Object_Type is Integer range 1 .. 5;`
Le tableau contient des `U_Object_Type` et a pour indice `U_Index_Type` de début et fin non encore déterminés
`type U_Type is array (U_Index_Type range <>) of U_Object_Type;`
Le tableau est déclaré avec in indice couvrant toutes les valeurs de `U_Index_Type` et est initialisé Directement.

```
   U_Table : U_Type(U_Index_Type'Range) := (- 2 => 2,
                                            - 1 => 1,
                                              0 => 5,
                                              1 => 3,
                                              2 => 4,
                                              3 => 2);
```

8.5.2 Instantiation du générique pour le premier type de données

Il faut utiliser le début de la déclaration du générique:

```
type Object_Type is private;
type Index_Type is (<>);
type Object_Array_Type is array (Index_Type range <>) of Object_Type;
with function ">" (X : in Object_Type;
                   Y : in     Object_Type) return Boolean is <>;
with function "<" (X : in Object_Type;
```

Chaque ligne doit assossier les types réellements utilisés à ceux indeterminés contenus dans le code générique

- `type Object_Type is private;`

  C'est le type de l'objet stock dans le tableau: Object_Array_Type => U_Type

- `type Index_Type is (<>);`

  C'est le type de l'index du tableau: Index_Type => U_Index_Type

- `type Object_Array_Type is array (Index_Type range <>) of Object_Type;`

  C'est le type de tableau: Object_Array_Type => U_Type

- ```
  with function ">" (X : in Object_Type;
  Y : in     Object_Type) return Boolean is <>;
  ```

  C'est la fonction logique > pour Object_Type, si elle existe par défaut (c'est le cas ici). On l'indique au compilateur : ">" => ">"

- ```
  with function "<" (X : in Object_Type;
   Y : in     Object_Type) return Boolean is <>;
  ```

  C'est la fonction logique < pour Object_Type, si elle existe par défaut (c'est le cas ici).On l'indique au compilateur : "<" => "<"

La syntaxe complète:

```
package U_Bornes is new Demo(Object_Type       => U_Object_Type,
                             Index_Type        => U_Index_Type,
                             Object_Array_Type => U_Type,
                             ">"               => ">",
                             "<"               => "<");
```

8.5.3 Second type de données

L'index du tableau est une énumération:

```
type Fruit_Type is (Orange, Apple, Pear, Cherry);
```

Le contenu du tableau est un entier

Le tableau contient des `integer` et a pour indice `Fruit_Type` de début et fin non encore déterminés.

```
type Tableau_Type is array (Fruit_Type range <>) of Integer;
```

Le tableau est déclaré avec in indice couvrant toutes les valeurs de `Fruit_Type` et est initialisé Directement.

```
F_Table : Tableau_Type(Fruit_Type'Range) := (Orange => 25,
                                             Apple  => 4,
                                             Pear   => 8,
                                             Cherry => 1);
```

8.5.4 Instantiation du générique pour le second type de données

Il faut utiliser le début de la déclaration du générique:

- `type Object_Type is private;`
- `type Index_Type is (<>);`
- `type Object_Array_Type is array (Index_Type range <>) of Object_Type;`
- `with function ">" (X : in Object_Type;`
  `Y : in Object_Type) return Boolean is <>;`
- `with function "<" (X : in Object_Type;`

Chaque ligne doit associer les types réellement utilisés à ceux indéterminés contenus dans le code générique

- `type Object_Type is private;`

  C'est le type de l'objet stock dans le tableau: Object_Array_Type => Integer

- `type Index_Type is (<>);`

  C'est le type de l'index du tableau: Index_Type => Fruit_Type

- `type Object_Array_Type is array (Index_Type range <>) of Object_Type;`

  C'est le type de tableau: Object_Array_Type => Tableau_Type

- ```
  with function ">" (X : in Object_Type;
   Y : in     Object_Type) return Boolean is <>;
  ```

  C'est la fonction logique > pour Object_Type, si elle existe par défaut (c'est le cas ici). On l'indique au compilateur : ">" => ">"

- ```
  with function "<" (X : in Object_Type;
   Y : in     Object_Type) return Boolean is <>;
  ```

  C'est la fonction logique < pour Object_Type, si elle existe par défaut (c'est le cas ici).On l'indique au compilateur : "<" => "<"

La syntaxe complète:

```
package F_Bornes is new Demo(Object_Type       => Integer,
                             Index_Type        => Fruit_Type,
                             Object_Array_Type => Tableau_Type,
                             ">"               => ">",
                             "<"               => "<");
```

8.5.5 Le code du programme principal

Il est dans tdemo.adb

```
with Demo;
```

```
with Ada.Text_Io;

procedure Tdemo
 is


 subtype U_Index_Type is Integer range - 2 .. 3;
 subtype U_Object_Type is Integer range 1 .. 5;

 type U_Type is array (U_Index_Type range <>) of U_Object_Type;

 U_Table : U_Type(U_Index_Type'Range) := (- 2 => 2,
                                          - 1 => 1,
                                          0   => 5,
                                          1   => 3,
                                          2   => 4,
                                          3   => 2);

 package U_Bornes is new Demo(Object_Type       => U_Object_Type,
                              Index_Type        => U_Index_Type,
                              Object_Array_Type => U_Type,
                              ">"               => ">",
                              "<"               => "<");


 type Fruit_Type is (Orange, Apple, Pear, Cherry);
 type Tableau_Type is array (Fruit_Type range <>) of Integer;

 F_Table : Tableau_Type(Fruit_Type'Range) := (Orange => 25,
                                              Apple  => 4,
                                              Pear   => 8,
                                              Cherry => 1);

 package F_Bornes is new Demo(Object_Type       => Integer,
                              Index_Type        => Fruit_Type,
                              Object_Array_Type => Tableau_Type,
                              ">"               => ">",
                              "<"               => "<");

 U_Min             : U_Object_Type;
 U_Max             : U_Object_Type;
 U_Value_At_Middle : U_Object_Type;
 U_Indice_Min      : U_Index_Type;
 U_Indice_Max      : U_Index_Type;
 U_Size            : Integer;


 begin

 U_Bornes.Bornes(Tableau         => U_Table,
                 Min             => U_Min,
                 Max             => U_Max,
                 Value_At_Middle => U_Value_At_Middle,
                 Indice_Min      => U_Indice_Min,
                 Indice_Max      => U_Indice_Max,
                 Size            => U_Size);

 Ada.Text_Io.Put_Line("U_Min "
                      & U_Object_Type'Image(U_Min));

 Ada.Text_Io.Put_Line("U_Max "
                      & U_Object_Type'Image(U_Max));

 Ada.Text_Io.Put_Line("U_Value_At_Middle "
                      & U_Object_Type'Image(U_Value_At_Middle));

 Ada.Text_Io.Put_Line("U_Indice_Min "
                      & U_Index_Type'Image(U_Indice_Min));

 Ada.Text_Io.Put_Line("U_Indice_Max "
                      & U_Index_Type'Image(U_Indice_Max));

 Ada.Text_Io.Put_Line("U_Size "
                      & Integer'Image(U_Size));


 Ada.Text_Io.New_Line;

 declare

 type Fruit_Type is (Orange, Apple, Pear, Cherry);
 type Tableau_Type is array (Fruit_Type range <>) of Integer;

 F_Table : Tableau_Type(Fruit_Type'Range) := (Orange => 25,
                                              Apple  => 4,
                                              Pear   => 8,
                                              Cherry => 1);

 package F_Bornes is new Demo(Object_Type       => Integer,
                              Index_Type        => Fruit_Type,
                              Object_Array_Type => Tableau_Type,
                              ">"               => ">",
                              "<"               => "<");



 F_Min             : Integer;
```

```
   F_Max            : Integer;
   F_Value_At_Middle : Integer;
   F_Indice_Min      : Fruit_Type;
   F_Indice_Max      : Fruit_Type;
   F_Size            : Integer;

   begin


     F_Bornes.Bornes(Tableau         => F_Table,
                     Min             => F_Min,
                     Max             => F_Max,
                     Value_At_Middle => F_Value_At_Middle,
                     Indice_Min      => F_Indice_Min,
                     Indice_Max      => F_Indice_Max,
                     Size            => F_Size);

     Ada.Text_Io.Put_Line("F_Min "
                          & Integer'Image(F_Min));

     Ada.Text_Io.Put_Line("F_Max  "
                          & Integer'Image(F_Max));

     Ada.Text_Io.Put_Line("F_Value_At_Middle  "
                          & Integer'Image(F_Value_At_Middle));

     Ada.Text_Io.Put_Line("F_Indice_Min "
                          & Fruit_Type'Image(F_Indice_Min));

     Ada.Text_Io.Put_Line("F_Indice_Max "
                          & Fruit_Type'Image(F_Indice_Max));

     Ada.Text_Io.Put_Line("F_Size "
                          & Integer'Image(F_Size));
   end;

  end Tdemo;
```

8.5.6 Le résultat du programme principal

```
F:\ada\current\w\miage>tdemo
U_Min  1
U_Max  5
U_Value_At_Middle  2
U_Indice_Min -1
U_Indice_Max  0
U_Size  6

F_Min  1
F_Max   25
F_Value_At_Middle   8
F_Indice_Min CHERRY
F_Indice_Max ORANGE
F_Size  4
F:\ada\current\w\miage>
```

8.6 LES FONCTIONS ET PROCÉDURES DONT UN GÉNÉRIQUE A BESOIN

Dans l'exemple précèdent les données ont les opérateurs de comparaison définis par défaut.
Ce n'est pas toujours le cas. Complétons l'exemple précèdent par un tableau contenant nom et prénom, défini comme suit:

```
type String_Access_Type is access all String;

   type Record_Type is record
     First_Name : String_Access_Type := null;
     Last_Name  : String_Access_Type := null;
   end record;

   subtype U_Index_Type is Integer range 1 .. 5;

   type U_Type is array (U_Index_Type range <>) of Record_Type;

   U_Table : U_Type(U_Index_Type'Range) := (1          =>
                                           (First_Name => new String'("Andre"),
                                            Last_Name  => new String'("Abadie")),
                                           2          =>
                                           (First_Name => new String'("Daniel"),
                                            Last_Name  => new String'("Levi")),
                                           3          =>
                                           (First_Name => new String'("David"),
                                            Last_Name  => new String'("Levi")),
                                           4          =>
                                           (First_Name => new String'("Daniel"),
                                            Last_Name  => new String'("Dupont")),
                                           5          =>
                                           (First_Name => new String'("David"),
                                            Last_Name  => new String'("Dupont")));
```

Les fonctions logiques ne sont plus définies pour `Record_Type`.

8.7 LES FONCTIONS LOGIQUES: ÉGAL SUPÉRIEUR ET INFÉRIEURE

Dans ce cas il faut créer les deux fonctions correspondantes décrivant comment classer un nom et un prénom. Par nom puis par prénom ou l'inverse!!

8.7.1 La fonction <

```
--|
 --|              |__________|
 --|              | Less_Than |
 --|              |__________|
 --|
 function Less_Than(Left  : in     Record_Type;
                    Right : in     Record_Type)
   return Boolean
   is
   Answer : Boolean := False;
   begin
   --|
   --|  check if Last_Name are equal on both side
   --|
   if Left.Last_Name.all = Right.Last_Name.all
     then
     --|
     --|   Last_Name are equal use First_Name
     --|
     Answer := Left.First_Name.all < Right.First_Name.all;
     else
     --|
     --|   Last_Name are different use Last_Name
     --|
     Answer := Left.Last_Name.all < Right.Last_Name.all;
   end if;

   return Answer;
 end Less_Than;
```

8.7.2 La fonction >

```
 --|
 --|              |__________|
 --|              |More_Than |
 --|              |__________|
 --|
 function More_Than(Left  : in     Record_Type;
                    Right : in     Record_Type)
   return Boolean
   is
   Answer : Boolean := False;
   begin
   --|
   --|  check if Last_Name are equal on both side
   --|
   if Left.Last_Name.all = Right.Last_Name.all
     then
     --|
     --|   Last_Name are equal use First_Name
     --|
     Answer := Left.First_Name.all > Right.First_Name.all;
     else
     --|
     --|   Last_Name are different use Last_Name
     --|
     Answer := Left.Last_Name.all > Right.Last_Name.all;
   end if;

   return Answer;
 end More_Than;
```

8.7.3 Instantiation du générique

Tout a fait semblable mais les fonctions > et < sont explicitement définies:

```
package U_Bornes is new Demo(Object_Type       => Record_Type,
                             Index_Type        => U_Index_Type,
                             Object_Array_Type => U_Type,
                             ">"               => More_Than,
                             "<"               => Less_Than);
```

8.7.4 programme principal

Dans le fichier tdemo_1.adb

```
with Demo;
  with Ada.Text_Io;
  procedure Tdemo_1
   is
   type String_Access_Type is access all String;

   type Record_Type is record
     First_Name : String_Access_Type := null;
     Last_Name  : String_Access_Type := null;
   end record;

   subtype U_Index_Type is Integer range 1 .. 5;

   type U_Type is array (U_Index_Type range <>) of Record_Type;

   U_Table : U_Type(U_Index_Type'Range) := (1          =>
                                           (First_Name => new String'("Andre"),
                                            Last_Name  => new String'("Abadie")),
                                           2          =>
                                           (First_Name => new String'("Daniel"),
                                            Last_Name  => new String'("Levi")),
                                           3          =>
                                           (First_Name => new String'("David"),
                                            Last_Name  => new String'("Levi")),
                                           4          =>
                                           (First_Name => new String'("Daniel"),
                                            Last_Name  => new String'("Dupont")),
                                           5          =>
                                           (First_Name => new String'("David"),
                                            Last_Name  => new String'("Dupont")));

   --|
   --|              |____________|
   --|              | Less_Than  |
   --|              |____________|
   --|
   function Less_Than(Left  : in     Record_Type;
                      Right : in     Record_Type)
     return Boolean
     is
     Answer : Boolean := False;
     begin
     --|
     --|  check if Last_Name are equal on both side
     --|
     if Left.Last_Name.all = Right.Last_Name.all
       then
       --|
       --|   Last_Name are equal use First_Name
       --|
       Answer := Left.First_Name.all < Right.First_Name.all;
       else
       --|
       --|   Last_Name are different use Last_Name
       --|
       Answer := Left.Last_Name.all < Right.Last_Name.all;
     end if;

     return Answer;
   end Less_Than;
   --|
   --|              |___________|
   --|              |More_Than  |
   --|              |___________|
   --|
   function More_Than(Left  : in     Record_Type;
                      Right : in     Record_Type)
     return Boolean
     is
     Answer : Boolean := False;
     begin
     --|
     --|  check if Last_Name are equal on both side
     --|
     if Left.Last_Name.all = Right.Last_Name.all
       then
       --|
       --|   Last_Name are equal use First_Name
       --|
       Answer := Left.First_Name.all > Right.First_Name.all;
       else
       --|
       --|   Last_Name are different use Last_Name
       --|
       Answer := Left.Last_Name.all > Right.Last_Name.all;
     end if;

     return Answer;
   end More_Than;

   package U_Bornes is new Demo(Object_Type       => Record_Type,
                                Index_Type        => U_Index_Type,
```

```
                              Object_Array_Type => U_Type,
                              ">"               => More_Than,
                              "<"               => Less_Than);


   U_Min             : Record_Type;
   U_Max             : Record_Type;
   U_Value_At_Middle : Record_Type;
   U_Indice_Min      : U_Index_Type;
   U_Indice_Max      : U_Index_Type;
   U_Size            : Integer;

   begin

   U_Bornes.Bornes(Tableau         => U_Table,
                   Min             => U_Min,
                   Max             => U_Max,
                   Value_At_Middle => U_Value_At_Middle,
                   Indice_Min      => U_Indice_Min,
                   Indice_Max      => U_Indice_Max,
                   Size            => U_Size);

   Ada.Text_Io.Put_Line("U_Min "
                        & U_Min.First_Name.all
                        & " "
                        & U_Min.Last_Name.all);

   Ada.Text_Io.Put_Line("U_Max "
                        & U_Max.First_Name.all
                        & " "
                        & U_Max.Last_Name.all);


   Ada.Text_Io.Put_Line("U_Value_At_Middle "
                        & U_Value_At_Middle.First_Name.all
                        & " "
                        & U_Value_At_Middle.Last_Name.all);

   Ada.Text_Io.Put_Line("U_Indice_Min "
                        & U_Index_Type'Image(U_Indice_Min));

   Ada.Text_Io.Put_Line("U_Indice_Max "
                        & U_Index_Type'Image(U_Indice_Max));

   Ada.Text_Io.Put_Line("U_Size "
                        & Integer'Image(U_Size));


   Ada.Text_Io.New_Line;


  end Tdemo_1;
```

8.7.5 résultat du programme principal

Voici l'écran correspondant

```
F:\ada\current\w\miage>gnatmake tdemo_1
gcc -c tdemo_1.adb
gcc -c demo.adb
gnatbind -x tdemo_1.ali
gnatlink tdemo_1.ali

F:\ada\current\w\miage>tdemo_1
U_Min Andre Abadie
U_Max David Levi
U_Value_At_Middle Daniel Levi
U_Indice_Min  1
U_Indice_Max  3
U_Size  5
F:\ada\current\w\miage>
```

8.8 AUTRES EXEMPLES

8.8.1 Utilisation de générique déjà écrits: Un exemple de tri commenté

Nous allons mettre en œuvre un tri générique (c'est un exemple très courant). Voici l'extrait du code correspondant à la déclaration:

```
 type Data_Type  is private;
```

```
 type Index_Type is (<>);
 type Array_Type is array (Index_Type range <>) of Data_Type;
 with function "<" (Left  : in Data_Type;
                    Right : in Data_Type)
 return Boolean is <>;
```

Tableau d'entier indice entier

Tri d'un tableau d'entiers avec un indice entier.

```
subtype I_Index_Type is Integer range 1 .. 10;
   type I_Array_Type is array (I_Index_Type range <>) of Integer;
```

L'instantiation du générique suit les règles vues plus haut:

- Indice du tableau Index_Type => I_Index_Type
- Contenu du tableau Data_Type => Integer
- Type de tableau Array_Type => I_Array_Type

L'instruction complète:

```
   package Si is new Sort(Index_Type => I_Index_Type,
                          Data_Type  => Integer,
                          Array_Type => I_Array_Type);
```

Tableau de réels indice énumération

Tri d'un tableau de réels avec un indice énumération.

```
type E_Index_Type is (Vert, Jaune, Rouge, Bleu, Noir, Blanc, Magenta, Cyan);
type E_Array_Type is array (E_Index_Type range <>) of Float;
```

L'instantiation du générique suit les règles vues plus haut:

- Indice du tableau Index_Type => E_Index_Type
- Contenu du tableau Data_Type => Float
- Type de tableau Array_Type => E_Array_Type

L'instruction complète:

```
      package Se is new Sort(Index_Type => E_Index_Type,
                             Data_Type  => Float,
                             Array_Type => E_Array_Type);
```

8.8.2 Fichier t.adb

Le code complet du test:

```
with Sort;
 with Ada.Text_Io;
 procedure T
   is

   subtype I_Index_Type is Integer range 1 .. 10;

   type I_Array_Type is array (I_Index_Type range <>) of Integer;

   I_Array : I_Array_Type(I_Index_Type'Range);

   package Si is new Sort(Index_Type => I_Index_Type,
                          Data_Type  => Integer,
                          Array_Type => I_Array_Type);


   type E_Index_Type is (Vert, Jaune, Rouge, Bleu, Noir, Blanc, Magenta, Cyan);
   type E_Array_Type is array (E_Index_Type range <>) of Float;

   E_Array : E_Array_Type(E_Index_Type'Range);

   package Se is new Sort(Index_Type => E_Index_Type,
                          Data_Type  => Float,
                          Array_Type => E_Array_Type);



   begin

   I_Array := (9, 8, 7, 5, 6, 4, 3, 1, 0, 2);

   Si.Shell(I_Array);

   for K in I_Array'Range loop

     Ada.Text_Io.Put_Line("k"
                          & I_Index_Type'Image(K)
                          & " I_Array(K)"
                          & Integer'Image(I_Array(K)));
   end loop;
```

```
   Ada.Text_Io.New_Line(Spacing => 2);

   E_Array := (- 1.1, 7.0, 3.2, 4.9, 3.3, 2.5, 3.8, 99.0);

   Se.Shell(E_Array);

   for K in E_Array'Range loop

     Ada.Text_Io.Put_Line("  k "
                          & E_Index_Type'Image(K)
                          & " I_Array(K)"
                          & Float'Image(E_Array(K)));

   end loop;


end T;
```

8.8.3 fichier sort.ads

```
-- shell sort from SOFTWARE COMPONENTS WITH ADA, GRADY BOOCH 1986 page 466,....
 generic
 type Data_Type is private;
 type Index_Type is (<>);
 type Array_Type is array (Index_Type range <>) of Data_Type;
 with function "<" (Left  : in Data_Type;
                    Right : in     Data_Type)
 return Boolean is <>;
 package Sort
 is
 procedure Shell(Data_Array : in out Array_Type);
 end Sort;
```

8.8.4 fichier sort.adb

```
-- shell sort from SOFTWARE COMPONENTS WITH ADA, GRADY BOOCH 1986 page 466,....
 package body Sort
   is
   procedure Shell(Data_Array : in out Array_Type)
     is
     Outer_Index : Index_Type;
     Dummy_Data  : Data_Type;
     Inner_Index : Index_Type;
     Increment   : Positive := 1;
     begin
     loop
       exit when (((9 * Increment) + 4) >= (Index_Type'Pos(Data_Array'Last)
                         - Index_Type'Pos(Data_Array'First)
                         + 1));
       Increment := 1 + (3 * Increment);
     end loop;
     loop
       Outer_Index := Index_Type'Val(Index_Type'Pos(Data_Array'First) + Increment);
       loop
         Dummy_Data  := Data_Array(Outer_Index);
         Inner_Index := Outer_Index;
         while
          Dummy_Data
          <
           Data_Array(Index_Type'Val(Index_Type'Pos(Inner_Index)- increment))
          loop
          Data_Array(Inner_Index) :=Data_Array(Index_Type'Val(Index_Type'Pos(Inner_Index)
                                     - Increment));
           Inner_Index := Index_Type'Val(Index_Type'Pos(Inner_Index) - Increment);
           exit when (Index_Type'Pos(Inner_Index) - Increment
                     < Index_Type'Pos(Data_Array'First));
         end loop;
         Data_Array(Inner_Index) := Dummy_Data;
         exit when ((Index_Type'Pos(Outer_Index) + Increment)
                   > index_Type'Pos(Data_Array'Last));
         Outer_Index := Index_Type'Val(Index_Type'Pos(Outer_Index) + Increment);
       end loop;
       exit when Increment = 1;
       Increment := (Increment - 1) / 3;
     end loop;
   end Shell;
 end Sort;
```

8.8.5 Le résultat écran

```
F:\ada\today>t
k 1 I_Array(K) 0
k 2 I_Array(K) 1
k 3 I_Array(K) 2
```

```
k 4 I_Array(K) 3
k 5 I_Array(K) 4
k 6 I_Array(K) 5
k 7 I_Array(K) 6
k 8 I_Array(K) 7
k 9 I_Array(K) 8
k 10 I_Array(K) 9


  k VERT I_Array(K)-1.10000E+00
  k JAUNE I_Array(K) 2.50000E+00
  k ROUGE I_Array(K) 3.20000E+00
  k BLEU I_Array(K) 3.30000E+00
  k NOIR I_Array(K) 3.80000E+00
  k BLANC I_Array(K) 4.90000E+00
  k MAGENTA I_Array(K) 7.00000E+00
  k CYAN I_Array(K) 9.90000E+01

F:\ada\today>
```

9 PROGRAMMATION OBJET

9.1 LE TYPE 'TAGGED'

Exemple de déclaration de coordonnées x y:

```
type Point is tagged record
X, Y : float := 0.0;
end record;
```

exemple de déclaration vide pouvant être utilisée directement.

```
type Expression is tagged null record;
```

exemple de déclaration vide NE pouvant PAS être utilisée directement: aucun objet de ce type ne peut être créé il servira d'ancêtre pour tous les types hérités.

```
type Data_Type is abstract tagged null record;
```

et le pointeur associé pour pouvoir lire les données stockées

```
type Data_Access_Type is access all Data_Type'Class;
```

fonction sans code (body) associée devant être héritée et étendue

```
function init(data:  out Data_Type) is abstract;
```

9.2 L'EXTENSION D'UN TYPE 'TAGGED' ET HÉRITABILITÉ

Les composant x et y sont hérités du type ancestral:

```
type Painted_Point is new Point with record Paint : Color := White; end record; -
```

exemple de declaration et d'initialisation d'un objet de ce type

```
Origin : constant Painted_Point := (X | Y => 0.0, Paint => Black);
```

une feuille d'un arbre représentant une expression arithmétique contient un opérande

```
type Literal is new Expression with record
Value : float;
end record;
```

et le pointeur associé (pour construire l'arbre)

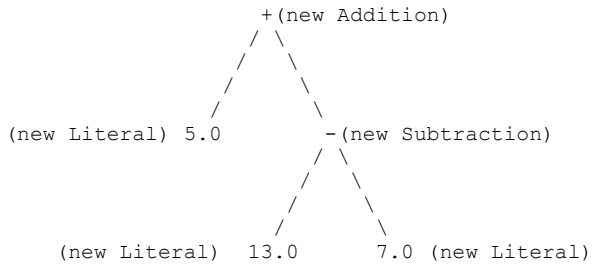
```
type Expr_Ptr is access all Expression'Class; _


type Binary_Operation is new Expression with record
Left, Right : Expr_Ptr;
end record;
```

Aucun composant supplémentaire n'est nécessaire pour les opérateurs ici + et -

```
type Addition is new Binary_Operation with null record;
type Subtraction is new Binary_Operation with null record;


Tree : Expr_Ptr :=
new Addition'( Left => new Literal'(Value => 5.0),
Right => new Subtraction'( Left => new Literal'(Value => 13.0),
```

```
Right => new Literal'(Value => 7.0)));
```

l'arbre construit ci-dessus représentant 5.0+(13.0-7.0) se dessine (remarquer le stockage dans un même objet, ici un arbre d'abstractions différentes: opérateurs et opérandes) les parenthèses sont intégrées pendant le codage en arbre:



Une possibilité de calcul peut être écrite comme suit:

```
function Calcul(T : in     Tagged_Expression_Tree.Expr_Ptr)
    return Float
    is
    begin
    if T.all in Tagged_Expression_Tree.Literal'Class
       then
       return Tagged_Expression_Tree.Literal(T.all).Value;
       elsif T.all in Tagged_Expression_Tree.Addition'Class
       then
       return Calcul(Tagged_Expression_Tree.Addition(T.all).Left)
       + Calcul(Tagged_Expression_Tree.Addition(T.all).Right);
       elsif T.all in Tagged_Expression_Tree.Subtraction'Class
       then
       return Calcul(Tagged_Expression_Tree.Subtraction(T.all).Left)
       - Calcul(Tagged_Expression_Tree.Subtraction(T.all).Right);
    end if;
    return 0.0;
end Calcul;
```

9.3 LE POLYMORPHISME EN UTILISANT LE CONCEPT DE 'D*ISPATCHING OPERATIONS'*

Pour cet exemple nous reprendrons le code du calculateur ci dessus avec utilisation du dispatching.

```
package Tagged_Expression_Tree
   is
  --exemple de déclaration vide pouvant être utilisée directement.
  -- elle servira pour d'finir tous les types feuilles et de noeuds de l'arbre

   type Expression is abstract tagged null record;

  --et le pointeur associé (pour construire l'arbre)

   type Expr_Ptr is access all Expression'Class;

  -- sa fonction associée est "abstract" : elle servira
  -- pour toutes les extensions et le 'dispatching automatique'.

   function Do_Calcul(T : in     Expression) return Float is abstract;

  -- les opérandes ont deux opérateurs, il faut définir une extension
  -- avec une branche gauche et une branche droite en utilisant deux pointeurs

   type Binary_Operation is new Expression with record
      Left, Right : Expr_Ptr;
   end record;

  -- et sa fonction associée

   function Do_Calcul(T : in     Binary_Operation) return Float;

  -- Aucun composant supplémentaire n'est nécessaire pour les opérateurs ici + , - ,/ et *
  -- ils sont définis pour utiliser le dispatching à travers les fonctions associées

   type Addition is new Binary_Operation with null record;
   type Subtraction is new Binary_Operation with null record;
   type Multiplication is new Binary_Operation with null record;
   type Division is new Binary_Operation with null record;
```

```
  -- et leurs fonctions associées

   function Do_Calcul(T : in     Addition) return Float;
   function Do_Calcul(T : in     Subtraction) return Float;
   function Do_Calcul(T : in     Multiplication) return Float;
   function Do_Calcul(T : in     Division) return Float;

  -- une feuille d'un arbre repr?sentant une expression arithmétique contient un opérande:
  -- il manque une extension pour les nombres;

   type Literal is new Expression with record
      Value : Float;
   end record;

  -- et sa fonction associée

   function Do_Calcul(T : in     Literal) return Float;

  --                  E X E M P L E   D ' A R B R E
  --                  =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=

  --    Tree : Expr_Ptr :=
  --    new Addition'(Left => new Literal'(Value => 5.0),
  --                  Right  =>
  --                  new Subtraction'(Left  =>
  --                                   new Literal'(Value => 13.0),
  --                                   Right  => new Literal'(Value => 7.0)));
  --
  -- l'arbre construit ci-dessus représentant  5.0+(13.0-7.0) se dessine (remarquer le stockage dans un même objet,
  -- ici un arbre d'abstractions différentes: opérateurs et opérandes).
  -- Il faut remarquer que les opérateur sont stockés sous forme d'un type!!!!
  --   les parenthèses sont intégrées pendant le codage en arbre:
  --  postsfix 5.0 13.0 7.0 - 6 *  et parcours post order de l'arbre
  --
   --                      +(new Addition)
   --                            / \
   --                           /   \
   --                          /     \
   --                         /       \
   --          (new Literal) 5.0   -(new Subtraction)
   --                                    / \
   --                                   /   \
   --                                  /     \
   --                                 /       \
   --              (new Literal)  13.0       7.0 (new Literal)

end Tagged_Expression_Tree;
```

L'observation du fonctionnement avec un "debugger" symbolique permet de voir tous les détails du fonctionnement.
Voici les codes:
Fichier : Tagged_Expression_Tree.adb

```
package body Tagged_Expression_Tree
  is
  --
  -- fonction utilise pour le 'dispatching' de +
  --
  function Do_Calcul(T : in     Addition)
     return Float
     is
     begin
     return Do_Calcul(T.Left.all) + Do_Calcul(T.Right.all);
     --        dispatching               dispatching
  end Do_Calcul;
  --
  -- fonction utilise pour le 'dispatching' de /
  --
  function Do_Calcul(T : in     Division)
     return Float
     is
     begin
     return Do_Calcul(T.Left.all) / Do_Calcul(T.Right.all);
     --        dispatching               dispatching
  end Do_Calcul;
  --
  -- fonction utilise pour le 'dispatching' de litteral
  --
  function Do_Calcul(T : in     Binary_Operation)
     return Float
     is
     begin
     return Do_Calcul(T);
     -- dispatching  vers l'une des fonctions precedentes
  end Do_Calcul;
```

```
end Tagged_Expression_Tree;
```

fichier T .adb

```
with Tagged_Expression_Tree;
with Ada.Text_Io;

procedure T
   is

   package My_Float_Io is new Ada.Text_Io.Float_Io(Float);

   Tree : Tagged_Expression_Tree.Expr_Ptr :=
   new Tagged_Expression_Tree.Addition'(Left  => new Tagged_Expression_Tree.Literal'(Value => 5.0),
                                        Right =>
                                        new Tagged_Expression_Tree.Subtraction'
                                        (Left                                =>
                                         new Tagged_Expression_Tree.Literal'(Value => 13.0),
                                         Right                               =>
                                         new Tagged_Expression_Tree.Literal'(Value => 7.0)));

   -- l'arbre construit ci-dessus repr@sentant  5.0+(13.0-7.0) se dessine (remarquer le stockage dans un m@me objet,
   -- ici un arbre d'abstractions diff@rentes: op@rateurs et op@randes) les parenth@ses sont int@gr@es pendant le codage en
arbre:
   -- postsfix 5.0 13.0 - 7.0 6 *  et parcours post order de l'arbre
   --
   --                              +(new Addition)
   --                             / \
   --                            /   \
   --                           /     \
   --                          /       \
   --         (new Literal) 5.0         -(new Subtraction)
   --                                   / \
   --                                  /   \
   --                                 /     \
   --                                /       \
   --             (new Literal)  13.0       7.0 (new Literal)

   --parcours post order le l'arbre binaire avec pointeurs:

   Data : Float;
   begin

   Data := Tagged_Expression_Tree.Do_Calcul(T => Tree.all);

   My_Float_Io.Put(Item => Data,
                   Fore => 5,
                   Aft  => 2,
                   Exp  => 0);
end T;
```

9.4 EXEMPLE DE DISPATCHING POUR LE CALCUL DES SURFACES DE TRIANGLES

La surface d'un triangle dont la longueur des cotés est a,b et c peut se calculer comme suit:
soit P = 1/2(a+b+c), la surface S= (p(p-a)(p-b)(p-c))**0.5. Différents types de triangles existent : équilatéral, isocèle, rectangle et quelconques. Une filiation peut être établie:

1. équilatéral a=b=c
2. isocèle a=b et c≠ a
3. rectangle a≠b, b≠c et c= $\sqrt{(a^2+b^2)}$
4. quelconques a≠b, b≠c et a≠c

cette filiation se retrouve dans les extensions du type de base:

Fichier triangle.ads:

```
package Triangle
   is

  -- La surface d'un triangle de cotés de longueur a, b et c est:
  -- soit P = 1/2(a+b+c)
  -- s= (p(p-a)(p-b)(p-c))**0.5

   type Triangle_Type is abstract tagged null record;
   type Pointeur_Vers_Triangle is access all Triangle_Type'Class;
  -- sa fonction associée est "abstract" : elle servira
  -- pour toutes les extensions et le 'dispatching automatique'.
   function Surface_Du_Triangle(T : in     Triangle_Type) return Float is abstract;
   procedure Lecture(T            : in out Triangle_Type) is abstract;
 -- une seule longueur de coté
   type Triangle_Equilateral is new Triangle_Type with record
      Longueur_Du_Cote_1 : Float;
   end record;
   function Surface_Du_Triangle(T : in     Triangle_Equilateral) return Float;
```

```
   procedure Lecture(T            : in out Triangle_Equilateral);
 -- deux longueurs de coté
   type Triangle_Isocele is new Triangle_Type with record
      Longueur_Du_Cote_1 : Float;
      Longueur_Du_Cote_2 : Float;
   end record;
   function Surface_Du_Triangle(T : in     Triangle_Isocele) return Float;
   procedure Lecture(T            : in out Triangle_Isocele);
 -- deux longueurs de coté aussi mais le calcul est différent
   type Triangle_Rectangle is new Triangle_Isocele with null record;
   function Surface_Du_Triangle(T : in     Triangle_Rectangle) return Float;
   procedure Lecture(T            : in out Triangle_Rectangle);
 -- trois longueurs de cotés
   type Triangle_Simple is new Triangle_Rectangle with record
      Longueur_Du_Cote_3 : Float;
   end record;
   function Surface_Du_Triangle(T : in     Triangle_Simple) return Float;
   procedure Lecture(T            : in out Triangle_Simple);
end Triangle;
```

Fichier triangle.adb

```
with Ada.Numerics.Generic_Elementary_Functions;
--  sin log exp racine puissance
with Ada.Text_Io;

package body Triangle
   is
   package Math is new Ada.Numerics.Generic_Elementary_Functions(Float_Type => Float);

   function "**" (X : in Float;Y : in Float)
      return Float
   renames Math."**";
  -- La surface d'un triangle de cotes de longueur a, b et c     est calculée comme suit:
  -- soit P = 1/2(a+b+c)
  -- s= (p(p-a)(p-b)(p-c))**0.5
  -- LE MÊME CALCUL EST UTILISE  POUR TOUS LES CAS

   function Surface_Du_Triangle(T : in     Triangle_Equilateral)
      return Float
      is
      A, B, C : Float := T.Longueur_Du_Cote_1;
      P       : Float := 0.5 * (A + B + C);
      begin
      return (P * (P - A) * (P - B) * (P - C)) ** 0.5;
   end Surface_Du_Triangle;



   function Surface_Du_Triangle(T : in     Triangle_Isocele)
      return Float
      is
      A, B : Float := T.Longueur_Du_Cote_1;
      C    : Float := T.Longueur_Du_Cote_2;
      P    : Float := 0.5 * (A + B + C);
      begin
      return (P * (P - A) * (P - B) * (P - C)) ** 0.5;
   end Surface_Du_Triangle;



   function Surface_Du_Triangle(T : in     Triangle_Rectangle)
      return Float
      is
      A : Float := T.Longueur_Du_Cote_1;
      B : Float := T.Longueur_Du_Cote_2;
      C : Float := (T.Longueur_Du_Cote_1 ** 2 + T.Longueur_Du_Cote_2 ** 2) ** 0.5;
      P : Float := 0.5 * (A + B + C);
      begin
      return (P * (P - A) * (P - B) * (P - C)) ** 0.5;
   end Surface_Du_Triangle;



   function Surface_Du_Triangle(T : in     Triangle_Simple)
      return Float
      is
      A : Float := T.Longueur_Du_Cote_1;
      B : Float := T.Longueur_Du_Cote_2;
      C : Float := T.Longueur_Du_Cote_3;
      P : Float := 0.5 * (A + B + C);
      begin
      return (P * (P - A) * (P - B) * (P - C)) ** 0.5;
   end Surface_Du_Triangle;



   procedure Lecture(T : in out Triangle_Equilateral)
      is
      E : String(1 .. 10);
      I : Positive;
```

```
      begin
      Ada.Text_Io.Put_Line(" triangle equilateral, entrer la longueur du cote ");
      E := (others              => ' ');
      Ada.Text_Io.Get_Line(Item => E,
                           Last => I);
      T.Longueur_Du_Cote_1 := Float'Value(E(E'First .. I));
   end Lecture;

   procedure Lecture(T : in out Triangle_Isocele)
      is
      E_1, E_2 : String(1 .. 10);
      I        : Positive;
      begin
      Ada.Text_Io.Put_Line(" triangle isocèle, entrer la longueur des deux cotes identiques ");
      E_1 := (others            => ' ');
      Ada.Text_Io.Get_Line(Item => E_1,
                           Last => I);
      T.Longueur_Du_Cote_1 := Float'Value(E_1(E_1'First .. I));

      Ada.Text_Io.Put_Line(" triangle isocèle, entrer la longueur du cote different ");
      E_2 := (others            => ' ');
      Ada.Text_Io.Get_Line(Item => E_2,
                           Last => I);
      T.Longueur_Du_Cote_2 := Float'Value(E_2(E_2'First .. I));

   end Lecture;

   procedure Lecture(T : in out Triangle_Rectangle)
      is
      E_1, E_2 : String(1 .. 10);
      I        : Positive;
      begin
      Ada.Text_Io.Put_Line(" triangle rectangle, entrer la longueur du cote angle gauche   ");
      E_1 := (others            => ' ');
      Ada.Text_Io.Get_Line(Item => E_1,
                           Last => I);
      T.Longueur_Du_Cote_1 := Float'Value(E_1(E_1'First .. I));

      Ada.Text_Io.Put_Line(" triangle rectangle, entrer la longueur du du cote angle droit ");
      E_2 := (others            => ' ');
      Ada.Text_Io.Get_Line(Item => E_2,
                           Last => I);
      T.Longueur_Du_Cote_2 := Float'Value(E_2(E_2'First .. I));
   end Lecture;

   procedure Lecture(T : in out Triangle_Simple)
      is
      E_1, E_2, E_3 : String(1 .. 10);
      I             : Positive;
      begin
      Ada.Text_Io.Put_Line(" triangle quelconque, entrer la longueur du premier cote ");
      E_1 := (others            => ' ');
      Ada.Text_Io.Get_Line(Item => E_1,
                           Last => I);
      T.Longueur_Du_Cote_1 := Float'Value(E_1(E_1'First .. I));

      Ada.Text_Io.Put_Line(" triangle quelconque, entrer la longueur du second cote ");
      E_2 := (others            => ' ');
      Ada.Text_Io.Get_Line(Item => E_2,
                           Last => I);
      T.Longueur_Du_Cote_2 := Float'Value(E_2(E_2'First .. I));

      Ada.Text_Io.Put_Line(" triangle quelconque, entrer la longueur du troisième cote ");
      E_3 := (others            => ' ');
      Ada.Text_Io.Get_Line(Item => E_3,
                           Last => I);
      T.Longueur_Du_Cote_3 := Float'Value(E_3(E_3'First .. I));
   end Lecture;


end Triangle;
```

Fichier Test_Triangle.adb

```
with Ada.Text_Io;
with Triangle;

procedure Test_Triangle
   is
   package My_Float_Io is new Ada.Text_Io.Float_Io(Float);

   Entree        : String(1 .. 100)               := (others => ' ');
   Longueur      : Natural                        := Natural'First;
   Pointeur      : Triangle.Pointeur_Vers_Triangle := null;
   Triangle_Equi : Triangle.Triangle_Equilateral;
   Triangle_Iso  : Triangle.Triangle_Isocele;
   Triangle_Rect : Triangle.Triangle_Rectangle;
   Triangle_Sim  : Triangle.Triangle_Simple;
   Surface       : Float;
   begin

   Ada.Text_Io.Put_Line("choix : 1=  equilateral, 2=  isocele, 3= rectangle , 4=quelquonque");
   Ada.Text_Io.Get_Line(Entree, Longueur);

   case Entree(Entree'First)
      is
```

```
      when '1'=>
      Pointeur := new Triangle.Triangle_Equilateral'(Triangle_Equi);

      when '2'=>
      Pointeur := new Triangle.Triangle_Isocele'(Triangle_Iso);

      when '3'=>
      Pointeur := new Triangle.Triangle_Rectangle'(Triangle_Rect);

      when '4'=>
      Pointeur := new Triangle.Triangle_Simple'(Triangle_Sim);

      when others =>
      Ada.Text_Io.Put(" entree inconnue");
      return;
   end case;

  -- lecture des donnees avec "dispatching"
   Triangle.Lecture(T => Pointeur.all);
  -- calcul de la surface avec "dispatching"
   Surface := Triangle.Surface_Du_Triangle(T => Pointeur.all);
  -- impression écran avec mise en forme
   My_Float_Io.Put(Item => Surface,
                   Fore => 5,
                   Aft  => 2,
                   Exp  => 0);
end Test_Triangle;
```

9.5 CALCUL D'UNE EXPRESSION ARITHMÉTIQUE POSTSCRIPT EN UTILISANT UN ARBRE D'EXPRESSIONS ET UN STACK

C'est une réutilisation du code d'un exemple précédent. Cette fois l'arbre est construit directement à partir de l'expression en postscript et un stack (voir le code du stack dans ce cours).

```
with Tagged_Expression_Tree;
with Ada.Text_Io;
with Simple_Stack;


procedure T
   is
-- declaration d'un stack de pointeurs vers les noeuds et feuilles de l'arbre
   package Stack is new Simple_Stack(Data_Type         => Tagged_Expression_Tree.Expr_Ptr,
                                     Stack_Size        => 10);
   package My_Float_Io is new Ada.Text_Io.Float_Io(Num => Float);
   subtype Str is String(1 .. 10);
   type Chain is array (1  .. 13) of Str;
-- chaîne contient les opérateurs et opérandes
   Chaine       : Chain                          := (others => (others => ' '));
   Node_Pointer : Tagged_Expression_Tree.Expr_Ptr := null;
   A            : Tagged_Expression_Tree.Expr_Ptr := null;
   B            : Tagged_Expression_Tree.Expr_Ptr := null;
   Tree         : Tagged_Expression_Tree.Expr_Ptr := null;
   Data         : Float                          := 0.0;
   S            : Stack.Stack_Type;

   begin
  -- toujours initialiser une structure avant de s'en servir
   Stack.Initialize(The_Stack => S);


  -- 2*3/(2-1)+5*(4-1)     2 3 * 2 1 - / 5 4 1 - * +    ces données sont codées ci-dessous dans chaine

   Chaine := (01 => "2" & (2 .. Str'Last => ' '),
              02 => "3" & (2 .. Str'Last => ' '),
              03 => "*" & (2 .. Str'Last => ' '),
              04 => "2" & (2 .. Str'Last => ' '),
              05 => "1" & (2 .. Str'Last => ' '),
              06 => "-" & (2 .. Str'Last => ' '),
              07 => "/" & (2 .. Str'Last => ' '),
              08 => "5" & (2 .. Str'Last => ' '),
              09 => "4" & (2 .. Str'Last => ' '),
              10 => "1" & (2 .. Str'Last => ' '),
              11 => "-" & (2 .. Str'Last => ' '),
              12 => "*" & (2 .. Str'Last => ' '),
              13 => "+" & (2 .. Str'Last => ' '));


   for I in Chaine'Range loop
      case Chaine(I) (1)
         is
   --    A L G O R I T H M E  D E  T R A D U C T I O N
   --    P O S T S C R I P T  =>  A R B R E  D ' E X P R E S S I O N
```

```
   -- boucle sur toute l'expression
   --si operande
   -- créer une feuille, la remplir avec la valeur et la mettre dans le stack
   -- si operateur:
   --lire le stack dans A
   --pop stack
   --lire le stack dans b
   --pop stack
   --créer un noeud du type correspondant à l'operateur avec:
   --                                  à gauche B
   --                                  à droite A
   --le mettre dans le stack
   --fin boucle
   -- l'arbre est dans le stack

       when '+'=>

   -- lecture A (pointeur vers le sous arbre de ce coté)

       A := Stack.Top_Of(The_Stack => S);
       Stack.Pop(The_Stack         => S);

   -- lecture B (pointeur vers le sous arbre de ce coté)

       B := Stack.Top_Of(The_Stack => S);
       Stack.Pop(The_Stack         => S);

   -- creation du nouveau pointeur vers le noeud correspondant à l'operateur
   -- avec B a gauche et A a droite

       Node_Pointer := new Tagged_Expression_Tree.Addition'(Left  => B,
                                                            Right => A);
   -- mettre dans le stack

       Stack.Push(Data         => Node_Pointer,
                  In_The_Stack => S);
   -- mêmes commentaires que pour l'opérateur précédent
       when '-'=>

       A := Stack.Top_Of(The_Stack => S);
       Stack.Pop(The_Stack         => S);

       B := Stack.Top_Of(The_Stack => S);
       Stack.Pop(The_Stack         => S);

       Node_Pointer := new Tagged_Expression_Tree.Subtraction'(Left  => B,
                                                               Right => A);

       Stack.Push(Data         => Node_Pointer,
                  In_The_Stack => S);
       -- mêmes commentaires que pour l'opérateur précédent
       when '/'=>

       A := Stack.Top_Of(The_Stack => S);
       Stack.Pop(The_Stack         => S);

       B := Stack.Top_Of(The_Stack => S);
       Stack.Pop(The_Stack         => S);

       Node_Pointer := new Tagged_Expression_Tree.Division'(Left  => B,
                                                            Right => A);

       Stack.Push(Data         => Node_Pointer,
                  In_The_Stack => S);
       -- mêmes commentaires que pour l'opérateur précédent
       when '*'=>

       A := Stack.Top_Of(The_Stack => S);
       Stack.Pop(The_Stack         => S);

       B := Stack.Top_Of(The_Stack => S);
       Stack.Pop(The_Stack         => S);

       Node_Pointer := new Tagged_Expression_Tree.Multiplication'(Left  => B,
                                                                  Right => A);

       Stack.Push(Data         => Node_Pointer,
                  In_The_Stack => S);

       when others =>
   -- c'est un operande créer une feuille, la remplir avec la valeur float
   -- et la mettre le pointeur dans le stack
       Node_Pointer := new Tagged_Expression_Tree.Literal'(Value => Float'Value(Chaine(I)));

       Stack.Push(Data         => Node_Pointer,
                  In_The_Stack => S);
    end case;
 end loop;
```

```
   -- lire dans le stack le pointeur vers le somment de l'arbre
    Tree := Stack.Top_Of(The_Stack => S);
   --faire le calcul
    Data := Tagged_Expression_Tree.Do_Calcul(T => Tree.all);
   -- affichage ecran
    My_Float_Io.Put(Item => Data,
                    Fore => 5,
                    Aft  => 2,
                    Exp  => 0);
end T;
```

10 TASK

10.1 INTRODUCTION

Dans la plupart des projets, il existe des parties de code qui sont indépendantes et peuvent s'exécuter en même temps. Le langage Ada a prévu cette possibilité sous forme de "tasks". Ces parties indépendantes, s'exécutent simultanément. Elles peuvent communiquer entre elles, s'attendre à des rendez-vous pour synchroniser leur action, partager des données de manière sécurisée (protected body) , etc.

10.2 UN PREMIER EXEMPLE: ÉCRITURE SUR L'ÉCRAN PAR DEUX TÂCHES NON COORDONNÉES

Dans cet exemple nous allons illustrer d'un manière très simple le fonctionnement d'un programme principal et de deux tâches s'exécutant simultanément au programme principal. Nous avons donc trois parties de code s'exécutant simultanément. Voici le résultat pour le programme task_1:

```
[dg@newdg miage_2005_2006]$ ./task_1
the A's are from task 1, the B's from task 2 and the spaces from the main
 task 1 started
A task 2 started
BABAB BABABA BABABA BABABA BAB ABA BBABAA BBBAA ABBB AAABB BAAA BBBAAA BBB AAAB BBAAA BBBAAA BBBA
AAB BBAAA BBBAAA BBBA AABBB AAA BBBA AABBB AAABBB ABABAB A BABAB ABABAB  task 2 Terminated
 task 1 Terminated
[dg@newdg miage_2005_2006]$
```

On observe l'apparition aléatoire des espaces, des A et des B.

10.2.1 Le programme principal

10.2.1.1 Définition des tâches

Les tâches sont définies comme suit dans le fichier task_1_A.Ads, pour simplifier il n'y a qu'une seule entrée sans paramètre.

```
 task type Example_Of_Task_1_Task_Type
      is
Une entrée est comme une procédure dans un package (ici un task)
      entry Start;
   end Example_Of_Task_1_Task_Type;

task type Example_Of_Task_2_Task_Type
      is
      entry Start;
   end Example_Of_Task_1_Task_Type;
```

Pour simplifier encore davantage les codes des deux tâches sont identiques (sauf la lettre imprimée).

10.2.1.2 Activation des tâches

Les tâches sont activées dés leur déclaration dans task_1_A.Ads (plusieurs tâches de même type peuvent être "créées" , il existe des possibilités de création dynamique avec pointeurs,.....)

```
Example_Of_Task_1_Task : Example_Of_Task_1_Task_Type;
Example_Of_Task_2_Task : Example_Of_Task_2_Task_Type;
```

10.2.1.3 Démarrage et arrêt des tâches

```
procedure Run_Tasks
      is
      begin
```
Pour simplifier la communication entre le programme principal et les tâches, la communication se fait par un drapeau logique en variable commune, des méthodes plus sophistiquée existent mais en introduction nous utiliseront une méthode simpliste.
```
      --|
      --| reset exit flag
      --|
      Task_1_A.Exit_The_Task := False;
```
L'activation est faite à la déclaration, le démarrage se fait par l'appel de l'entrée. Il existe des moyens de vérifier que la tâche est toujours active par des "chien de garde,...)
```
      --|
      --|start task number 1
      --|
      Task_1_A.Example_Of_Task_1_Task.Start;
      --|
      --|start task number 2
      --|
      Task_1_A.Example_Of_Task_2_Task.Start;
```
Le programme principal fonctionne ici en parallèle avec les deux tâches et imprime des espaces avec une boucle simulant un travail pendant 30 secondes.
```
      --|
      --|Simulate something going on here for 30 seconds
      --|
      for I in 1 .. 30 loop
         delay 1.0;
         Ada.Text_Io.Put(' ');
      end loop;
```
Ici le drapeau est mis à vrai pour communiquer aux deux tâches de s'achever à la prochaine exécution de la boucle.
```
      --|
      --| set exit flag => both tasks will exit
      --|
      Task_1_A.Exit_The_Task := True;
```
Il faut attendre que la boucle des deux tâches ait le temps de s'exécuter
```
      delay 3.0;-- wait for tasks to terminate
   end Run_Tasks;
```

10.2.2 La tâche numéro 1

10.2.2.1 Les différentes parties d'une tâche

```
   task body Example_Of_Task_1_Task_Type
      is
      begin
```
C'est là la boucle sur toutes les entrées définies pour cette tâche
```
      loop
         select
            accept Start
               do
              .....
```
*Le code situé ici est exécuté **AVANT** de rendre la main au programme appelant cette entrée. Attention, les variables déclarées ici ou reçus en paramètre ne sont **PLUS** visibles dans la partie simultanée ci-dessous!*
```
            end Start;

            or
            terminate;
         end select;
```
Nom de la partie simultanée
```
         Scroll_When_Mouse_On_Frame :
         declare
```
Partie déclarative pour le code exécuté simultanément avec le programme principal
```
         begin
```
*Le code situé ici est exécuté **UNE FOIS** **parallèlement** avec le programme principal et les autres tâches*
```
            loop
```
La boucle permet une exécution aussi longue que possible ce code étant exécuté une seule fois
```
            end loop;
```
Fin de la partie simultanée
```
         end Scroll_When_Mouse_On_Frame;
      end loop;
   end Example_Of_Task_1_Task_Type;
```

10.2.2.2 La partie s'exécutant seule

```
accept Start
               do
               --|
               --|-----------------------------------------|
               --| this is where the code of the task is   |
               --| executed in exclusion to anything else  |
               --|-----------------------------------------|
               --|

               Ada.Text_Io.Put_Line(" task 1 started");
            end Start;
```

10.2.2.3 La partie s'exécutant simultanément avec le programme principal

Nom de cette partie

```
Scroll_When_Mouse_On_Frame :
         declare
```

Partie déclarative (ici vide mais c'est parce que le programme est simpliste)

```
         begin
```

Partie exécutée une fois en parallèle la boucle permet de fonctionner aussi longtemps que nécessaire

```
            loop
```

Sorte de boucle et fin de la tâche dés que le drapeau est vrai

```
               exit when Exit_The_Task;
```

Impression de 3 lettres A simulant du temps de traitement

```
               Ada.Text_Io.Put("A");
               delay 0.1;
               Ada.Text_Io.Put("A");
               delay 0.1;
               Ada.Text_Io.Put("A");
               delay 0.1;
```

Simulation d'un traitement

```

               delay 1.0;
            end loop;
```

Fin de la tâche avec impression écran

```
            Ada.Text_Io.Put_Line(" task 1 Terminated");
```

*Une exception est **INDISPENSABLE** pour les tâches qui peuvent se terminer sans message pendant que le programme principal et les autres tâches se poursuivent. Dans un programme réel il faut un message et un traitement approprié!*

```
            exception
            when Info : others => null;
```

Fin de la partie simultanée

```
         end Scroll_When_Mouse_On_Frame;
```

10.2.3 La tâche numéro 2

Le code est le même sauf que La tâche numéro 2 imprime des B.

```
Ada.Text_Io.Put("A");
delay 0.1;
 Ada.Text_Io.Put("B");
 delay 0.1;
 Ada.Text_Io.Put("B");
 delay 0.1;
```

10.3 SECOND EXEMPLE : UTILISATION D'UN SÉMAPHORE POUR COORDONNER LES DEUX TÂCHES

10.3.1 Introduction

L'absence totale de synchronisation observée ci-dessus ne permet qu'une utilisation simpliste des tâches. Bien entendu, une grande variété de méthodes existent pour synchroniser une ou plusieurs tâches et un programme principal. Le plus simple est le sémaphore.

10.3.2 But d'un sémaphore

Un sémaphore permet à une tâche de bloquer le fonctionnement de l'autre tâche. Il possède deux routines seize et release. Seize permet au premier appelant de 'prendre la main' les tâches suivantes sont mises dans une queue et ne peuvent continuer leur exécution. Release est l'opération inverse. Pour éviter tout appel de size ou relase avant la fin de leur execution (OS multi-tâche préemptif) le langage ada à prévu la notion de 'protected'.L'utilisation de protected oblige un appel à se terminer avant l'exécution de l'appel suivant. Il ne peut donc pas y avoir deux appels de plusieurs tâches s'exécutant simultanément. Les opérations seize et release sont protégées et ne peuvent s'exécuter simultanément.

10.3.2.1 Seize

Cette routine permet à la première tâche qui appelle de continuer son fonctionnement. Les tâches suivantes qui appellent son mises en attente dans une QUEUE et ne s'exécutent plus.

10.3.2.2 Release

Cette routine permet à la tâche qui a pris le contrôle de le relâcher permettant ainsi à la tâche suivante de la queue de s'exécuter

10.3.3 Le sémaphore écrit en protected type

```
   protected type Semaphore_Type
      is
Protected est utilisé pour empêcher un appel ré-entrant par le multi-tâche au milieu de l'exécution de seize et de release
      procedure Release;

      entry Seize;

      private
Défini waiting comme interne
      entry Waiting;
Permet d'identifier la tâche appelant seize, la même tâche ayant appelé avec succès seize pourra le rappeler sans être bloquée
      Owner : Ada.Task_Identification.Task_Id;
      Count : Natural := 0;
   end Semaphore_Type;


   protected body Semaphore_Type
      is

      procedure Release
         is
         begin
Le code est simple car une tâche bloquée ne peut appeler release
         Count := Count - 1;
      end Release;

      entry Seize when True
         is
         use type Ada.Task_Identification.Task_Id;
         begin
         if Owner = Seize'Caller
            then
            Count := Count + 1;

            else
            requeue Waiting with abort;

         end if;
      end Seize;

      entry Waiting when Count = 0
         is
         begin
         Count := 1;
         Owner := Waiting'Caller;
      end Waiting;

   end Semaphore_Type;
```

10.3.4 Fonctionnement du sémaphore

Release est simpliste mais il est "protected" il se déroule jusqu'à la fin excluant tout fonctionnement en parallèle de seize!!!.
L'appel de seize étant "protected" possède les mêmes propriétés de fonctionnement exclusif. Son déroulement est illustré ci-dessous:

```
entry Seize when True
         is
         use type Ada.Task_Identification.Task_Id;
         begin
1er appel le test est faux
1er appel  par une autre procédure/fonction le test est faux
Nième appel par la procédure/fonction dont le nom est enregistré le test est vrai: incrémentation de count. Il faudra donc appeler release autant de fois que seize pour remettre count à zéro.
         if Owner = Seize'Caller
            then
            Count := Count + 1;

            else
            requeue Waiting with abort;

         end if;
      end Seize;
1er appel count = 0 Le code s'exécute
1er appel  par une autre procédure/fonction: la barrière est fausse mis en attente dans une queue avec possibilité d' utiliser "abort" pour le programme appelant pour éviter un blockage si cette ressource n'est pas disponible (par exemple après un temps d'attente si la tâche contient:
select xxx(yyy);
or delay 7.0; -- délai avant abort
end select;

      entry Waiting when Count = 0
         is
         begin
         Count := 1;
         Owner := Waiting'Caller;
Count est 1 et owner est le nom du programme(procédure/fonction) appelant
      end Waiting;
```

10.3.5 Le même problème avec synchronisation par sémaphore

La sortie écran est:

2nd part: concurrency with semaphore
 task 1 started
A task 2 started
AABBB AAABBB AA ABBB AAABBB AAABBB AAA BBB AAABBB AA ABBB AAA BBBA AABBB AAABBB AAABB BAA ABBB AAABBB AAABBB AAA BBB AAABBB AAAB BBAAA BBB AAABBB AAABB BAA ABBB task 1 Terminated
 task 2 Terminated

On observe bien :
Les deux taches sont maintenant mutuellement exclusives car il y a toujours 3 A ou 3 B de suite. Les espaces en provenance du programme principal sont distribués au hasard car il n'est pas synchronisé.

10.4 TROISIÈME EXEMPLE: UTILISATION DE ADA.FINALIZATION POUR LA GESTION DU SÉMAPHORE

L'inconvénient majeur du système précédent est la **possibilité pour le programmeur d'oublier l'appel à la procédure release**.
Le language ada possède un outil qui permet de s'**assurer de l'appel de seize et release automatiquement**. Il est basé sur la création d'une variable et le moment où cette variable cesse d'exister:

```
procedure xxx ...
is
--x est créé
X: special;
begin
....
--x disparaît
end xxx;
```

...

L'outil Ada.Finalization permet d'associer un code à :

- La 'création' d'une variable
- La 'modification' de la valeur d'une variable
- La 'disparation' de la variable

10.4.1 Introduction à Ada.Finalization

Ce package propose les 3 procédures suivantes qui ne font rien par défaut mais peuvent être associées à des actions.

procedure Initialize (Object : **in out** Controlled);
procedure Adjust (Object : **in out** Controlled);
procedure Finalize (Object : **in out** Controlled);

Pour comprendre le fonctionnement un exemple simple utilisant initialize et finalize:

10.4.1.1 Fichier final.ads

```
package Final
   is
Création d'un type test_type qui est "protected"
   type Test_Type;
   type Access_Test_Type is access all Test_Type;
 Le "protected type" est vide et n'est associé à aucune action
   protected type Test_Type
      is
   end Test_Type;

end Final;
```

10.4.1.2 Fichier final.adb

```
Le fichier adb est vide (mais nécessaire au compilateur) car aucune action associée n'est présente
package body Final
   is
   protected body Test_Type
      is
   end Test_Type;
end Final;
```

10.4.1.3 Fichier final-protects.ads

```
Le code associe le type test_type à initialize et finalize
with Ada.Finalization;

package Final.Protects
   is

   type Example(Data : access Test_Type)
   is limited private;

   private

Null record car le type example est vide

   type Example(Data : access Test_Type)
   is new Ada.Finalization.Limited_Controlled
   with null record;

   procedure Initialize(Protect : in out Example);
   procedure Finalize(Protect   : in out Example);

end Final.Protects;
```

10.4.1.4 Fichier final-protects.adb

```
with Ada.Text_Io;
with System;

package body Final.Protects
   is
quant une variable de type example est initialisée cette procédure est appelée
   procedure Initialize(Protect : in out Example)
      is
      begin
      Ada.Text_Io.Put("INITIALIZE S");
   end Initialize;
quant une variable de type example est détruite cette procédure est appelée
   procedure Finalize(Protect : in out Example)
      is
      begin
      Ada.Text_Io.Put("FINALIZE S");
   end Finalize;

end Final.Protects;
```

10.4.1.5 Fichier t_final.adb

Le programme principal ne fait rien!! pourtant il imprime INITIALIZE S FINALIZE S sur l'écran

```
with Final;
with Final.Protects;

procedure T_Final
   is
   S : aliased Final.Test_Type;
l'initialisation de z appelle A U T O M A T I Q U E M E N T initialize
   Z : Final.Protects.Example(Data => S'Access);
   begin
   null;
la "destruction" de z appelle A U T O M A T I Q U E M E N T finalize
end T_Final;
```

10.4.2 Utilisation pour la gestion du sémaphore

L'idée illustrée ci dessus peut et va servir a appeler seize au moment de l'initialisation et release en fin de scope.

10.4.3 Le code avec Ada.Finalization

Le code est comme précédemment sauf pour l'appel à seize et à release qui est effectué par la variable z.

```
            loop

               exit when Exit_The_Task;
               Semaphore_Seize : declare
               S               : aliased Semaph.Semaphore_Type;
Seize est appelé automatiquement à l'initialisation de Z
               Z               : Semaph.Protects.Semaphore_Protect_Single(Semaphore => S'Access);
               begin
                  --|
                  --| SIMULATE HEAVY CALCULATIONS
                  --|
                  Ada.Text_Io.Put("A");
                  delay 0.1;
                  Ada.Text_Io.Put("A");
                  delay 0.1;
                  Ada.Text_Io.Put("A");
                  delay 0.1;

                  --|
                  --| simulate complex calculations
                  --|
                  delay 1.0;
release est appelé automatiquement à la fin du 'scope' de Z
               end Semaphore_Seize;
            end loop;
```

10.5 QUATRIÈME EXEMPLE: CRÉATION DYNAMIQUE DE PLUSIEURS TÂCHES

L'objet de cet exemple est de monter comment, pendant le fonctionnement d'un programme, on peut créer et détruire des tâches. Le calcul en parallèle de factorielle N, pour différentes valeurs de N, sera utilisé avec un code de tâche unique. En partant de ce code, on crée dynamiquement plusieurs fois cette tâche. Toutes les tâches ainsi crées fonctionnent en parallèle pour chaque calcul de factorielle.

10.5.1 Le discriminant d'une tâche

Lors de la création dynamique d'une tâche on peut transmettre à cette tâche nouvellement créée une ou plusieurs variables dont la valeur est STATIQUE et spécifique à cette nouvelle tâche ce qui permet de les différentier entre elles. Dans le cas présent il faut créer un nouveau discriminant pour chaque création d'une nouvelle tâche. C'est pourquoi un pointeur est utilisé.

Création d'un type record pour servir de discriminant, ici il est simpliste

```
type Data_Type is record
     N : Integer;
  end record;
```

Création d'un type pointeur pour le discriminant pour permettre sa création dynamique en même temps que la tâche associée.

```
   type Data_Access_Type is access all Data_Type;
```

Création d'un type task avec le discriminant associé

```
   task type Create_Task_Type(Data_Access : Data_Access_Type)  is ........
```

10.5.2 La déclaration d'une tâche en vue de sa création dynamique

Toute création dynamique d'un objet se fait en utilisant un pointeur et le mot clé NEW. Il faut donc définir un type à cet effet:

Déclaration du type Create_Task_Type avec un discriminant (data_access)

```
   task type Create_Task_Type(Data_Access : Data_Access_Type)
      is
```

Déclaration de la taille du stack associé à cette tâche

```
      pragma Storage_Size(12 * 1024);
```

Pour simplifier cette tâche n'a qu'une entrée

```
      entry Create(Task_Number : in     Integer);
   end Create_Task_Type;
```

Définition du type de pointeur associé au type de la tâche pour pouvoir créer des tâches de ce type dynamiquement

```
   type Create_Task_Access_Type is access Create_Task_Type;
```

10.5.3 L'appel d'une tâche avec discriminant

L'appel à la procédure calcul crée le pointeur qui sera initialisé pour le discriminant et la tâche

```
procedure Calcul(Number      : in     Positive;
                 Task_Number : in     Positive)
     is
     Create_Task_Access : Task_Global.Create_Task_Access_Type := null;
     Local_Data_Access  : Task_Global.Data_Access_Type        := null;
     Data               : Task_Global.Data_Type               := (N => Number);
```

10.5.4 Création dynamique d'une tâche

Cette création se fait en créant un nouvel objet à partir du type pointeur. On crée deux objets, le discriminant et la tâche:

Création de l'objet qui sert de discriminant (data est initialisé avant cette ligne)

```
      Local_Data_Access := new Task_Global.Data_Type'(Data);
```

Création de la tâche avec initialisation du discriminant en utilisant l'objet créé ci-dessus (qui sert de discriminant)

```
      Create_Task_Access := new Task_Global.Create_Task_Type(Data_Access => Local_Data_Access);
```

activation de la tâche avec un délai maximum autorisé pour cela:

```
      select
```

Activation de la tâche

```
         Create_Task_Access.Create(Task_Number => Task_Number);
         or
```

Spécification du délai maximum de création

```
         delay 0.2;
```

Action en cas de dépassement du délai maximum de création (tâche terminée par exemple)

```
         Ada.Text_Io.Put_Line("task not ready");
      end select;
```

10.5.5 Transmission de la variable d'appel à la partie de la tâche s'exécutanten parallèle et Utilisation du discriminant

```
with Ada.Text_Io;


package body Task_Global
   is
   task body Create_Task_Type
      is
```

La partie rouge s'exécute seule en exclusivité et ne rend la main que quant elle est terminée.
Dummy est une variable globale qui sert à transmettre la valeur de Task_Number entre la partie rouge et la partie verte. En effet le domaine d'existence de la variable Task_Number est limitée à la partie rouge.

```
      Dummy : Integer;
      begin
      loop
         select
            accept Create(Task_Number : in     Integer)
               do
               Dummy := Task_Number;
            end Create;
            or
            terminate;
         end select;
```

La partie verte s'exécute parallèlement à toutes les autres tâches et au programme principal appelant. Le calcul de la factorielle est simpliste mais permet au calcul le plus long de durer plus longtemps.

```
         Calcul_Of_Factorial :
         declare
```

Variable auxiliaire X initialisée en utilisant la valeur transmise par le discriminant contenu dans la variable data_access

```
         X      : Data_Access_Type := Data_Access;
         Result : Long_Float       := 1.0;
         begin
```

Impression écran immédiate du numéro de tâche

```
            Ada.Text_Io.Put_Line("task number "
                                 & Integer'Image(Dummy));
```

Calcul simpliste de N! En utilisant la valeur transmise par le discriminant contenu dans la variable data_access

```
            for I in Long_Integer(2) .. Long_Integer(X.all.N) loop
               Result := Result * Long_Float(I);
```

Rendre la main en cas d' "Operating sytem" non préemptif

```
               delay 0.01;
            end loop;
```

Impression écran du résultat du calcul

```
            Ada.Text_Io.Put_Line("factorial"
                                 & Integer'Image(X.all.N)
                                 & " ="
                                 & Long_Float'Image(Result));
```

Fin de la tâche et destruction automatique de celle-ci

```
         end Calcul_Of_Factorial;
      end loop;
   end Create_Task_Type;
end Task_Global;
```

10.5.6 Résultat du fonctionnement

Tout le code est disponible ci-dessous. Le résultat du fonctionnement illustre l'indépendance des tâches car la première tâche qui imprime les résultat est la dernière tâche créée. (Cet effet est créé volontairement pour illustrer le comportement).

```
with Call_Task;

procedure A
   is
   begin
   Call_Task.Calcul(Number      => 70,
                    Task_Number => 1);
   Call_Task.Calcul(Number      => 60,
                    Task_Number => 2);
   Call_Task.Calcul(Number      => 50,
                    Task_Number => 3);
   Call_Task.Calcul(Number      => 40,
                    Task_Number => 4);
   Call_Task.Calcul(Number      => 30,
                    Task_Number => 5);
   Call_Task.Calcul(Number      => 20,
                    Task_Number => 6);
   Call_Task.Calcul(Number      => 10,
                    Task_Number => 7);
end A;
```

```
[dg@newdg x]$ ./a
task number  1
task number  2
task number  3
task number  4
task number  5
task number  6
task number  7
factorial 10 = 3.62880000000000E+06
factorial 20 = 2.43290200817664E+18
factorial 30 = 2.65252859812191E+32
factorial 40 = 8.15915283247898E+47
factorial 50 = 3.04140932017134E+64
factorial 60 = 8.32098711274139E+81
factorial 70 = 1.19785716699699E+100
[dg@newdg x]$
```

10.6 CINQUIÈME EXEMPLE: LE DINNER DES PHILOSOPHES

C'est le plus grand classique d'actions simultanées et bloquantes.

Le texte ci-dessous vient de wikipedia

La situation est la suivante :

- Cinq philosophes (initialement mais il peut y en avoir beaucoup plus) entrent dans un restaurant chacun à leur tour et se trouvent autour d'une table **RONDE**.
- Chacun des philosophes a devant lui un plat de spaghetti.
- A gauche de chaque assiette se trouve une SEULE fourchette.

Un philosophe n'a que trois états possibles :

- penser pendant un temps indéterminé;
- être affamé (pendant un temps déterminé et fini sinon il y a famine) ;
- manger pendant un temps déterminé et fini.

Des contraintes extérieures s'imposent à cette situation :

- Si un philosophe a faim, il va entrer au restaurant si il y a une place à table. Ensuite il attend que les fourchettes gauche et droite soient libres.
- pour manger, un philosophe a besoin de deux fourchettes : celle qui se trouve à gauche de sa propre assiette, et celle qui se trouve à gauche de celle de son voisin de droite (c'est-à-dire les deux fourchettes qui entourent sa propre assiette) ;

- si un philosophe n'arrive pas à s'emparer d'une fourchette, il reste affamé pendant un temps déterminé, en attendant de renouveler sa tentative.

Le problème consiste à trouver un ordonnancement des philosophes tel qu'ils puissent tous manger, chacun à leur tour. Cet ordre est imposé par la solution que l'on considère .

Ici les temps d'attente sont tirés au sort.
Il y a plusieurs tâches:

- Une tâche par philosophe
- Une tâche par fourchette
- Une tâche représentant le restaurant.

Voici la partie principale commentée, le code complet est plus bas.

```
with Random_Normal;
with Ada.Text_Io;

package body Philosopher_Task
  is
 --|
 --| cette tache a pour but d'obliger une fourchette  PRECISE
 --| ( il y a 5 fourchettes) à être reposée sur la
 --| table avant de pourvoir être prise par une autre personne. cette tâche se termine
 --| quant Ada_PHILOSOPHE se termine
 --|
  task body Fork
     is
     begin
   --|
   --| boucle classique des taches en ada
   --|
     loop
        select
       --|
       --| entree prendre une fourchette     P R E C I S E
       --|                      ==============
           accept Pick_Up
              do
              Ada.Text_Io.Put_Line("    fourchette prise");
              null;
           end Pick_Up;
       --|
       --| entree reposer une fourchette     P R E C I S E
       --|                      ==============
           accept Put_Down
              do
              Ada.Text_Io.Put_Line("    fourchette posee");
              null;
           end Put_Down;
       --|
       --| suite syntaxe tache
       --|
           or
           terminate;
        end select;
     end loop;
  end Fork;
 --|
 --|    GESTION ENTREE SORTIE DES PHILOSOPHES dans le restaurant
 --|
  task body Host
     is
   --|
   --| global pour host
   --|
     I : Integer := 0;--number of philosophers in the room
     begin
     loop
        select
       --|
       --| test si entree possible
       --|
           when I < 4 =>
       --|
       --| ok
       --|
           accept Enter(Name : Philosopher_Data.Philosophe_Name)
              do
              Ada.Text_Io.Put_Line("Bonjour "
```

```
                                & Name);
            null;
         end Enter;
      --|
      --| nombre de philosophes
      --|
         I := I + 1;
         --|
         --|
         --|
         or
         --|
         --|
         --|
         accept Leave(Name : Philosopher_Data.Philosophe_Name)
            do
            Ada.Text_Io.Put_Line("AU revoir "
                                & Name);
            null;
         end Leave;
         --|
         --|nombre de philosophes
         --|
         I := I - 1;

         or
         terminate;
      end select;
   end loop;
end Host;
--|
--| GESTION DU philosophe
--|
task body Ada_Philosophe
   is
   --|
   --|
   --|
   I                : Philosopher_Data.Id;--numero du philosophe
   Max_Eats_Per_Day : constant := 4;
   Times_Eaten      : Integer  := 0;
   Left_Fork_Number : Philosopher_Data.Id;
   Right_Fork_Number: Philosopher_Data.Id;
   My_Name          : Philosopher_Data.Philosophe_Name;
   Random_Delay     : Long_Float;
   begin
  --|
  --| noter la syntaxe differente de cette tache
  --|
   accept Get_Id(J    : in     Philosopher_Data.Id;
                 Name : in     Philosopher_Data.Philosophe_Name)
      do
      --|
      --| j   disparaît aprés end Get_Id, il est mis en global pour la suite
      --|
      I  := J;
      --|
      --|
      --|
      Ada.Text_Io.Put_Line(My_Name
                           & " marche vers le restaurant");
   --|
   --| cette partie du code est en arrêt jusqu'à la fin du delai.
   --|
      Random_Delay := Random_Normal.Draw_Random(Mean              => 2.0,
                                                Standard_Deviation => 1.0,
                                                Max               => Long_Float'Last,
                                                Min               => 0.0);
      Ada.Text_Io.Put_Line(My_Name
                           & " attend pendant "
                           & Long_Float'Image(Random_Delay)
                           & " en marchant");
   --|
   --| attente
   --|
      delay Duration(Random_Delay);
   end Get_Id;
  --|
  --| les fourchettes sont numérotées à partir du numéro du
  --| philosophe (i):
  --|
  --|
  --|
  --|       PHILOSOPHE       FOURCHETTE GAUCHE      FOURCHETTE DROITE
  --|
  --|         1                         1              2
  --|         2                         2              3
  --|         3                         3              4
```

```
--|        4                              4             5
--|        5                              5             1
--|
--|
   Left_Fork_Number  := I;
   Right_Fork_Number := I mod 5 + 1;
--|
--|  le philosophe peut manger 4 fois
--|
   while Times_Eaten /= Max_Eats_Per_Day loop

      Ada.Text_Io.Put_Line(My_Name
                           & "discute avec son voisin ");
   --|
   --| calcul de l'attente avant d'entrer dans le restaurant
   --|
      Random_Delay := Random_Normal.Draw_Random(Mean               => 20.0,
                                                Standard_Deviation => 3.0,
                                                Max                => Long_Float'Last,
                                                Min                => 0.0);

      Ada.Text_Io.Put_Line(My_Name
                           & " lis le menu"
                           & Long_Float'Image(Random_Delay));
   --|
   --|attente avant d'entrer dans le restaurant
   --|
      delay Duration(Random_Delay);
   --|
   --| appel a une entree de tache: doit attendre dans la queue correspondante
   --| si elle n'est pas vide
   --|
      Host.Enter(My_Name);

      Ada.Text_Io.Put_Line(My_Name
                           & " entre et prend un siege ");
   --|
   --| appel a une entree de tache: doit attendre dans la queue correspondante
   --| si elle n'est pas vide
   --|
      Forks(Right_Fork_Number).Pick_Up;
      Ada.Text_Io.Put_Line(My_Name
                           & " prend sa fourchette droite ");
   --|
   --| appel a une entree de tache: doit attendre dans la queue correspondante
   --| si elle n'est pas vide
   --|
      Forks(Left_Fork_Number).Pick_Up;
      Ada.Text_Io.Put_Line(My_Name
                           & " prend sa fourchette droite ");
   --|
   --|calcul du temps pour manger
   --|
      Random_Delay := Random_Normal.Draw_Random(Mean               => 8.0,
                                                Standard_Deviation => 2.0,
                                                Max                => Long_Float'Last,
                                                Min                => 0.0);

      Ada.Text_Io.Put_Line(My_Name
                           & " va manger pendant "
                           & Long_Float'Image(Random_Delay));
   --|
   --|attente du temps pour manger
   --|
      delay Duration(Random_Delay);
   --|
   --| repas termine: poser la fourchette gauche : elle devient libre
   --| appel a une entree de tache: doit attendre dans la queue correspondante
   --| si elle n'est pas vide
   --|
      Forks(Left_Fork_Number).Put_Down;
      Ada.Text_Io.Put_Line(My_Name
                           & " a fini et repose sa fourchette gauche");
   --|
   --| repas termine: poser la fourchette droite : elle devient libre
   --|appel a une entree de tache: doit attendre dans la queue correspondante
   --| si elle n'est pas vide
   --|
      Forks(Right_Fork_Number).Put_Down;
      Ada.Text_Io.Put_Line(My_Name
                           & " repose sa fourchette droite");
   --|
```

```
      --| nombre de repas
      --|
         Times_Eaten := Times_Eaten + 1;

         Ada.Text_Io.Put_Line(My_Name
                              & " se leve et part");
      --|
      --|appel a une entree de tache: doit attendre dans la queue correspondante
      --| si elle n'est pas vide
      --|
         Host.Leave(My_Name);
      end loop;

      Ada.Text_Io.Put_Line(My_Name
                           & " va se coucher");
   -- c'est la fin de cette tache elle disparaît
  end Ada_Philosophe;

end Philosopher_Task;
```

10.7 LES CODES COMPLETS

10.8 PREMIER EXEMPLE

10.8.1 Task_1.adb

```
with Ada.Text_Io;-- this is a routine already available in ada for i/o of  t e x t (strings)
with Task_1_A;
with Ada.Task_Identification;
--|
--|                          |______|
--|                          |task_1|
--|                          |______|
procedure Task_1
   is
   procedure Run_Tasks
      is
      begin
      --|
      --| reset exit flag
      --|
      Task_1_A.Exit_The_Task := False;
      --|
      --|start task number 1
      --|
      Task_1_A.Example_Of_Task_1_Task.Start;
      --|
      --|start task number 2
      --|
      Task_1_A.Example_Of_Task_2_Task.Start;
      --|
      --|Simulate something going on here for 30 seconds
      --|
      for I in 1 .. 30 loop
         delay 1.0;
         Ada.Text_Io.Put(' ');
      end loop;
      --|
      --| set exit flag => both tasks will exit
      --|
      Task_1_A.Exit_The_Task := True;
      delay 3.0;-- wait for tasks to terminate
   end Run_Tasks;



   begin
   Ada.Text_Io.Put_Line("the A's are from task 1, the B's from task 2 and the spaces from the main");
   --|
   --| put header and start both tasks without semaphore
   --|

   Run_Tasks ;
   Ada.Text_Io.New_Line;
   --|
   --|is task number 1 still alive and running if so kill the task
   --|
   if Ada.Task_Identification.Is_Callable(T => Task_1_A.Example_Of_Task_1_Task'Identity)
      then
      abort Task_1_A.Example_Of_Task_1_Task;
   end if;
   --|
   --|is task number 2 still alive and running if so kill the task
   --|
   if Ada.Task_Identification.Is_Callable(T => Task_1_A.Example_Of_Task_2_Task'Identity)
      then
```

```
      abort Task_1_A.Example_Of_Task_2_Task;
   end if;

end Task_1;
```

10.8.2 Task_1_a.ads

```
package Task_1_A
   is
   --|
   --|                     T A S K S   D E C L A R A T I O N S
   --|        -=-=-=-

   --|
   --|                            Scroll_When_Mouse_On_Frame
   --|
   task type Example_Of_Task_1_Task_Type
      is
      entry Start;
   end Example_Of_Task_1_Task_Type;

   for Example_Of_Task_1_Task_Type'Storage_Size use 4 * 1024;

   --|
   --|                            Scroll_When_Mouse_On_Frame
   --|
   task type Example_Of_Task_2_Task_Type
      is
      entry Start;
   end Example_Of_Task_2_Task_Type;

   for Example_Of_Task_2_Task_Type'Storage_Size use 4 * 1024;

   --|
   --|tasks instantiation
   --|

   Example_Of_Task_1_Task : Example_Of_Task_1_Task_Type;
   Example_Of_Task_2_Task : Example_Of_Task_2_Task_Type;

   Exit_The_Task : Boolean := False;
end Task_1_A;
```

10.8.3 Task_1_a.adb

```
with Ada.Text_Io;
with Ada.Task_Identification;


package body Task_1_A
   is


   --|
   --|   |=================|
   --|   | Example_of_task_1     |
   --|   |=================|
   --|
   task body Example_Of_Task_1_Task_Type
      is

      --|
      --|  |=============================|
      --|  | M A I N     Example_of_task                |
      --|  |=============================|
      --|
      begin
      loop
         select
            accept Start
               do
               --|
               --|-----------------------------------------|
               --| this is where the code of the task is   |
               --| executed in exclusion to anything else  |
               --|-----------------------------------------|
               --|

               Ada.Text_Io.Put_Line(" task 1 started");
            end Start;
            or
            terminate;
         end select;
         Scroll_When_Mouse_On_Frame :
         declare


         begin
            --|
            --|-----------------------------------------|
            --| this is where the code of the task is   |
            --| executed AS AN INDEPENDENT PROCESS      |
```

```
         --|----------------------------------------|
         --|

         loop

            exit when Exit_The_Task;

            --|
            --| SIMULATE HEAVY CALCULATIONS
            --|
            Ada.Text_Io.Put("A");
            delay 0.1;
            Ada.Text_Io.Put("A");
            delay 0.1;
            Ada.Text_Io.Put("A");
            delay 0.1;

            --|
            --| simulate complex calculations
            --|
            delay 1.0;
         end loop;
         Ada.Text_Io.Put_Line(" task 1 Terminated");
         exception
         when Info : others => null;
      end Scroll_When_Mouse_On_Frame;
   end loop;
end Example_Of_Task_1_Task_Type;
--|
--|   |==================|
--|   | Example_of_task_2|
--|   |==================|
--|
task body Example_Of_Task_2_Task_Type
   is
   --|
   --|  |==============================|
   --|  | M A I N     Example_of_task  |
   --|  |==============================|
   --|
   begin
   loop
      select
         accept Start
            do
            --|
            --|----------------------------------------|
            --| this is where the code of the task is  |
            --| executed in exclusion to anything else |
            --|----------------------------------------|
            --|
            Ada.Text_Io.Put_Line(" task 2 started");
         end Start;
         or
         terminate;
      end select;
      Scroll_When_Mouse_On_Frame :
      declare


      begin
         --|
         --|----------------------------------------|
         --| this is where the code of the task is  |
         --| executed AS AN INDEPENDENT PROCESS     |
         --|----------------------------------------|
         --|

         loop
            exit when Exit_The_Task;

            --|
            --| put number on screen
            --|
            Ada.Text_Io.Put("B");
            delay 0.1;
            Ada.Text_Io.Put("B");
            delay 0.1;
            Ada.Text_Io.Put("B");
            delay 0.1;

            --|
            --| simulate complex calculations
            --|
            delay 0.95;
         end loop;
         Ada.Text_Io.Put_Line(" task 2 Terminated");

         exception
         when Info : others => null;
      end Scroll_When_Mouse_On_Frame;
   end loop;
end Example_Of_Task_2_Task_Type;
```

```
end Task_1_A;
```

10.9 SECOND EXEMPLE

10.9.1 Task_2.adb

```
with Ada.Text_Io;-- this is a routine already available in ada for i/o of  t e x t (strings)
with Task_2_A;
with Ada.Task_Identification;
--|                          |_______|
--|                          | task_2 |
--|                          |_______|
procedure Task_2
   is
   procedure Run_Tasks(Choice : in     Boolean)
      is
      begin
      --|
      --| reset exit flag
      --|
      Task_2_A.Exit_The_Task := False;
      --|
      --|start task number 1
      --|
      Task_2_A.Example_Of_Task_1_Task.Start(Use_The_Semaphore => Choice);
      --|
      --|start task number 2
      --|
      Task_2_A.Example_Of_Task_2_Task.Start(Use_The_Semaphore => Choice);
      --|
      --|Simulate something going on here for 30 seconds
      --|
      for I in 1 .. 30 loop
         delay 1.0;
         Ada.Text_Io.Put(' ');
      end loop;
      --|
      --| set exit flag => both tasks will exit
      --|
      Task_2_A.Exit_The_Task := True;
      delay 3.0;-- wait for tasks to terminate
   end Run_Tasks;


   begin
   Ada.Text_Io.Put_Line("the A's are from task 1, the B's from task 2 and the spaces from the main");
   --|
   --| put header and start both tasks without semaphore
   --|
   Ada.Text_Io.Put_Line("1st Part: concurrency WITHOUT  semaphore  ");
   Run_Tasks(Choice => False);
   Ada.Text_Io.New_Line;
   --|
   --| put header and start both tasks with semaphore
   --|
   Ada.Text_Io.Put_Line("2nd part: concurrency with  semaphore  ");
   Run_Tasks(Choice => True);
   --|
   --|is task number 1 still alive and running if so kill the task
   --|
   if Ada.Task_Identification.Is_Callable(T => Task_2_A.Example_Of_Task_2_Task'Identity)
      then
      abort Task_2_A.Example_Of_Task_2_Task;
   end if;
   --|
   --|is task number 1 still alive and running if so kill the task
   --|
   if Ada.Task_Identification.Is_Callable(T => Task_2_A.Example_Of_Task_2_Task'Identity)
      then
      abort Task_2_A.Example_Of_Task_2_Task;
   end if;

end Task_2;
```

10.9.2 Task_2_a.ads

```
package Task_2_A
   is
   --|
   --|                     T A S K S   D E C L A R A T I O N S
   --|        -=-=-=-

   --|
   --|                            Scroll_When_Mouse_On_Frame
   --|
   task type Example_Of_Task_1_Task_Type
```

```
      is
      entry Start(Use_The_Semaphore : in     Boolean := True);
   end Example_Of_Task_1_Task_Type;

   for Example_Of_Task_1_Task_Type'Storage_Size use 4 * 1024;

   --|
   --|                           Scroll_When_Mouse_On_Frame
   --|
   task type Example_Of_Task_2_Task_Type
      is
      entry Start(Use_The_Semaphore : in     Boolean := True);
   end Example_Of_Task_2_Task_Type;

   for Example_Of_Task_2_Task_Type'Storage_Size use 4 * 1024;

   --|
   --|tasks instantiation
   --|

   Example_Of_Task_1_Task : Example_Of_Task_1_Task_Type;
   Example_Of_Task_2_Task : Example_Of_Task_2_Task_Type;

   Exit_The_Task : Boolean := False;
end Task_2_A;
```

10.9.3 Task_2_a.adb

```
with Ada.Text_Io;
with Ada.Task_Identification;


package body Task_2_A
   is

   --|
   --|                Semaphore_Type
   --|                ==============
   --|
   --|
   --|a semaphore to avoid the simultaneous call of a procedure/function by two tasks .TO use from taskings-protects.ads
   --|
   type Semaphore_Type;
   type Access_Semaphore_Type is access all Semaphore_Type;

   protected type Semaphore_Type
      is

      procedure Release;

      entry Seize;

      private
      entry Waiting;
      Owner : Ada.Task_Identification.Task_Id;
      Count : Natural := 0;
   end Semaphore_Type;


   protected body Semaphore_Type
      is

      procedure Release
         is
         begin
         Count := Count - 1;
      end Release;

      entry Seize when True
         is
         use type Ada.Task_Identification.Task_Id;
         begin
         if Owner = Seize'Caller
            then
            Count := Count + 1;

            else
            requeue Waiting with abort;

         end if;
      end Seize;

      entry Waiting when Count = 0
         is
         begin
         Count := 1;
         Owner := Waiting'Caller;
      end Waiting;

   end Semaphore_Type;
   --|
   --|                          |====================|
   --|                          | GLOBAL SEMAPHORE   |
   --|                          |====================|
   --|
```

```
Semaphore : Semaphore_Type;
--|
--|    |==================|
--|    | Example_of_task_1|
--|    |==================|
--|
task body Example_Of_Task_1_Task_Type
   is
   Local_Use_The_Semaphore : Boolean := False;

   --|
   --|  |==============================|
   --|  | M A I N     Example_of_task  |
   --|  |==============================|
   --|
   begin
   loop
      select
         accept Start(Use_The_Semaphore : in     Boolean := True)
            do
            --|
            --|----------------------------------------|
            --| this is where the code of the task is  |
            --| executed in exclusion to anything else |
            --|----------------------------------------|
            --|
            Local_Use_The_Semaphore := Use_The_Semaphore;
            Ada.Text_Io.Put_Line(" task 1 started");
         end Start;
         or
         terminate;
      end select;
      Scroll_When_Mouse_On_Frame :
      declare


      begin
         --|
         --|----------------------------------------|
         --| this is where the code of the task is  |
         --| executed AS AN INDEPENDENT PROCESS     |
         --|----------------------------------------|
         --|

         loop

            exit when Exit_The_Task;
            --|
            --| wait until semaphore is free
            --|
            if Local_Use_The_Semaphore
               then
               Semaphore.Seize;
            end if;
            --|
            --| SIMULATE HEAVY CALCULATIONS
            --|
            Ada.Text_Io.Put("A");
            delay 0.1;
            Ada.Text_Io.Put("A");
            delay 0.1;
            Ada.Text_Io.Put("A");
            delay 0.1;
            --|
            --| release semaphore
            --|
            if Local_Use_The_Semaphore
               then
               Semaphore.Release;
            end if;
            --|
            --| simulate complex calculations
            --|
            delay 1.0;
         end loop;
         Ada.Text_Io.Put_Line(" task 1 Terminated");
         exception
         when Info : others => null;
      end Scroll_When_Mouse_On_Frame;
   end loop;
end Example_Of_Task_1_Task_Type;
--|
--|    |==================|
--|    | Example_of_task_2|
--|    |==================|
--|
task body Example_Of_Task_2_Task_Type
   is
   Local_Use_The_Semaphore : Boolean := False;
   --|
   --|  |==============================|
   --|  | M A I N     Example_of_task  |
   --|  |==============================|
   --|
   begin
   loop
      select
```

```
            accept Start(Use_The_Semaphore : in     Boolean := True)
               do
               --|
               --|----------------------------------------|
               --| this is where the code of the task is  |
               --| executed in exclusion to anything else |
               --|----------------------------------------|
               --|
               Local_Use_The_Semaphore := Use_The_Semaphore;
               Ada.Text_Io.Put_Line(" task 2 started");
            end Start;
            or
            terminate;
         end select;
         Scroll_When_Mouse_On_Frame :
         declare


         begin
            --|
            --|----------------------------------------|
            --| this is where the code of the task is  |
            --| executed AS AN INDEPENDENT PROCESS     |
            --|----------------------------------------|
            --|

            loop
               exit when Exit_The_Task;
               --|
               --| wait until semaphore is free
               --|
               if Local_Use_The_Semaphore
                  then
                  Semaphore.Seize;
               end if;
               --|
               --| put number on screen
               --|
               Ada.Text_Io.Put("B");
               delay 0.1;
               Ada.Text_Io.Put("B");
               delay 0.1;
               Ada.Text_Io.Put("B");
               delay 0.1;
               --|
               --| release semaphore
               --|
               if Local_Use_The_Semaphore
                  then
                  Semaphore.Release;
               end if;
               --|
               --| simulate complex calculations
               --|
               delay 0.95;
            end loop;
            Ada.Text_Io.Put_Line(" task 2 Terminated");

            exception
            when Info : others => null;
         end Scroll_When_Mouse_On_Frame;
      end loop;
   end Example_Of_Task_2_Task_Type;



end Task_2_A;
```

10.10 TROISIÈME EXEMPLE

10.10.1 Fichier semaph.ads

```
with Ada.Task_Identification;

package Semaph is

   pragma Elaborate_Body;

   --=====================================================================
   --| mandatory for compatibility with tagged types
   --| used in the BC components
   --| to be used as for example :
   --|
   --| TYPE DOUBLE_LIST IS NEW CONTAINER WITH RECORD
   --|    REP                : DOUBLE_NODES.DOUBLE_NODE_REF;
   --|    ACCESS_TO_SEMAPHORE : GUARD.PROTECTS.ACCESS_SEMAPHORE_TYPE
   --|   := NEW GUARD.SEMAPHORE_TYPE;
   --| END RECORD;
   --|
```

```
   --| please take note of the initialization !!!
   --====================================================================

   type Semaphore_Type;
   type Access_Semaphore_Type is access all Semaphore_Type;

   protected type Semaphore_Type is
      procedure Release;
      entry Seize;

      private
      entry Waiting;

      Owner : Ada.Task_Identification.Task_Id;
      Count : Natural := 0;

   end Semaphore_Type;

end Semaph;
```

10.10.2 Fichier semaph.adb

```
-- Version 1.0 July, 1999
-- Daniel.Gaudry@wanadoo.fr


with Ada.Text_Io;

package body Semaph
   is
   Debug : constant Boolean := True;

   use type Ada.Task_Identification.Task_Id;

   protected body Semaphore_Type
      is
      procedure Release
         is
         begin

         if Debug then
            Ada.Text_Io.Put_Line("  releasing  "
                                 & " # "
                                 & Integer'Image(Count));
         end if;

         Count := Count - 1;
      end Release;


      entry Seize
         when True
         is
         begin
         if Owner = Seize'Caller
            then

            Count := Count + 1;

            if Debug then
               Ada.Text_Io.Put_Line("  seizing : "
                                    & Ada.Task_Identification.Image(Seize'Caller)
                                    & " # "
                                    & Integer'Image(Count));
            end if;

            else

            if Debug then
               Ada.Text_Io.Put_Line("  REQUEUE WAITING : "
                                    & Ada.Task_Identification.Image(Seize'Caller)
                                    & " # "
                                    & Integer'Image(Count));
            end if;

            requeue Waiting with abort;

         end if;
      end Seize;

      entry Waiting
         when Count = 0
         is
         begin
         if Debug then
            Ada.Text_Io.Put_Line("  waiting : "
                                 & Ada.Task_Identification.Image(Waiting'Caller));
         end if;
         Count := 1;
         Owner := Waiting'Caller;
      end Waiting;

   end Semaphore_Type;
end Semaph;
```

10.10.3 fichier Semaph-Protects.ads

```
-- Version 1.0 July, 1999
-- Daniel.Gaudry@wanadoo.fr


--|this routine was written over the years from contributions from quite a few people among them
--|Original idea (Djistra)
--|Steve Carlsen
--|Mike Hayes
--|R.bruckard
--|Matthew Heaney
--|Yours Truly (double seize for procedures with two 'containers' as parameters
--|  and access_semaphore_type for compatibility of a protected type with tagged type

with Ada.Finalization;

package Semaph.Protects
   is

   pragma Elaborate_Body;

   --=====================================================================
   --| mandatory for compatibility with tagged types
   --| used in the BC components
   --| to be used as for example :
   --|
   --| TYPE DOUBLE_LIST IS NEW CONTAINER WITH RECORD
   --|    REP                : DOUBLE_NODES.DOUBLE_NODE_REF;
   --|    ACCESS_TO_SEMAPHORE : GUARD.PROTECTS.ACCESS_SEMAPHORE_TYPE
   --|   := NEW GUARD.SEMAPHORE_TYPE;
   --| END RECORD;
   --|
   --| please take note of the initialization !!!
   --=====================================================================


   --=====================================================================
   --|protect procedure/function with one 'container' in the parameter's
   --| list to be used as example :
   --|
   --|PROCEDURE CLEAR(OBJ : IN OUT DOUBLE_LIST) IS
   --|
   --|
   --|   X    : GUARD.PROTECTS.SEMAPHORE_PROTECT_SINGLE(SEMAPHORE => OBJ.ACCESS_TO_SEMAPHORE);
   --|   CURR : DOUBLE_NODES.DOUBLE_NODE_REF := OBJ.REP;
   --|   PTR  : DOUBLE_NODES.DOUBLE_NODE_REF;
   --|
   --| BEGIN
   --|.............
   --|
   --=====================================================================

   type Semaphore_Protect_Single(Semaphore : access Semaphore_Type)
   is limited private;

   --=====================================================================
   --|protect procedure/function with TWO 'containers' in the parameter's
   --| list to be used as example :
   --|
   --|FUNCTION "="(L, R : DOUBLE_LIST) RETURN BOOLEAN IS
   --|
   --| X_0            : GUARD.PROTECTS.SEMAPHORE_PROTECT_DOUBLE(SEMAPHORE_1 => L.ACCESS_TO_SEMAPHORE,
   --|                                                          SEMAPHORE_2 => R.ACCESS_TO_SEMAPHORE);
   --|BEGIN
   --|
   --|   RETURN L.REP = R.REP;
   --|END "=";
   --|
   --=====================================================================

   type Semaphore_Protect_Double(Semaphore_1 : access Semaphore_Type;
                                 Semaphore_2 : access Semaphore_Type)
   is limited private;


   private

   type Semaphore_Protect_Single(Semaphore : access Semaphore_Type)
   is new Ada.Finalization.Limited_Controlled with null record;

   type Semaphore_Protect_Double(Semaphore_1 : access Semaphore_Type;
                                 Semaphore_2 : access Semaphore_Type)
   is new Ada.Finalization.Limited_Controlled with null record;

   procedure Initialize(Protect : in out Semaphore_Protect_Single);
   procedure Finalize(Protect   : in out Semaphore_Protect_Single);

   procedure Initialize(Protect : in out Semaphore_Protect_Double);
   procedure Finalize(Protect   : in out Semaphore_Protect_Double);


end Semaph.Protects;
```

10.10.4 fichier Semaph-Protects.adb

```
-- Version 1.0 July, 1999
-- Daniel.Gaudry@wanadoo.fr


with Ada.Text_Io;
with System;

package body Semaph.Protects
   is

   --=====================================================================
   --|creating an object of type semaphore_control seizes the semaphore
   --=====================================================================

   procedure Initialize(Protect : in out Semaphore_Protect_Single) is
      begin
      Ada.Text_Io.Put("INITIALIZE S");

      Protect.Semaphore.Seize;
   end;

   procedure Initialize(Protect : in out Semaphore_Protect_Double) is
      use type System.Address;
      begin

      --=========================================================
      --| the key is to seize the semaphore of both 'containers'
      --| in the same order regardless of which is first
      --| in the event that two different tasks call this very
      --| procedure at the same time for the same two lists as for example :
      --| task1
      --|if (A=B) ...
      --| task2
      --| if (B=A) ...
      --| which is the most difficult case to care for
      --| the address is used to force the seize in the same order
      --| in both calls thereby preventing deadlock
      --=========================================================

      if Protect.Semaphore_1.all'Address < Protect.Semaphore_2.all'Address
         then

         Protect.Semaphore_1.Seize;
         Protect.Semaphore_2.Seize;

         else

         Protect.Semaphore_2.Seize;
         Protect.Semaphore_1.Seize;

      end if;

   end;

   --=====================================================================
   --| and at the end of its scope, the object is finalized and the semaphore released
   --| S I M P L E !!!
   --=====================================================================

   procedure Finalize(Protect : in out Semaphore_Protect_Single) is
      begin
      Ada.Text_Io.Put("FINALIZE S");

      Protect.Semaphore.Release;
   end;

   procedure Finalize(Protect : in out Semaphore_Protect_Double) is

      begin
      Ada.Text_Io.Put("FINALIZE D");

      Protect.Semaphore_1.Release;
      Protect.Semaphore_2.Release;
   end;

end Semaph.Protects;
```

10.10.5 fichier task_3_a.ads

```
--   --|Daniel.Gaudry@wanadoo.fr
--|
--|                                M_28_A
--|
--| Version 1.00 June         02 nd 2000 initial coding


package Task_3_A
   is
   --|
   --|                      T A S K S   D E C L A R A T I O N S
```

```
   --|        -=-=-=-

   --|
   --|                            Scroll_When_Mouse_On_Frame
   --|
   task type Example_Of_Task_1_Task_Type
      is
      pragma Storage_Size(4 * 1024);
      entry Start;
   end Example_Of_Task_1_Task_Type;

   --|
   --|                            Scroll_When_Mouse_On_Frame
   --|
   task type Example_Of_Task_2_Task_Type
      is
      entry Start;
      pragma Storage_Size(4 * 1024);
   end Example_Of_Task_2_Task_Type;

   --|
   --|tasks instantiation
   --|

   Example_Of_Task_1_Task : Example_Of_Task_1_Task_Type;
   Example_Of_Task_2_Task : Example_Of_Task_2_Task_Type;

   Exit_The_Task : Boolean := False;
end Task_3_A;
```

10.10.6 fichier task_3_a.adb

```
with Ada.Text_Io;
--with Ada.Task_Identification;
with Semaph.Protects;
with Semaph;
with Ada.Exceptions;
pragma Elaborate_All(Semaph.Protects);


package body Task_3_A
   is
   --|
   --|   |==================|
   --|   | Example_of_task_1|
   --|   |==================|
   --|
   task body Example_Of_Task_1_Task_Type
      is

      --|
      --|  |============================|
      --|  | M A I N     Example_of_task  |
      --|  |============================|
      --|
      begin
      loop
         select
            accept Start
               do
               --|
               --|----------------------------------------|
               --| this is where the code of the task is  |
               --| executed in exclusion to anything else |
               --|----------------------------------------|
               --|

               Ada.Text_Io.Put_Line(" task 1 started");
            end Start;
            or
            terminate;
         end select;
         Independent_Process :
         declare
         begin
            --|
            --|----------------------------------------|
            --| this is where the code of the task is  |
            --| executed AS AN INDEPENDENT PROCESS     |
            --|----------------------------------------|
            --|

            loop

               exit when Exit_The_Task;
               Semaphore_Seize : declare
               S               : aliased Semaph.Semaphore_Type;
               Z               : Semaph.Protects.Semaphore_Protect_Single(Semaphore => S'Access);
               begin
                  --|
                  --| SIMULATE HEAVY CALCULATIONS
                  --|
                  Ada.Text_Io.Put("A");
                  delay 0.1;
                  Ada.Text_Io.Put("A");
```

```
                delay 0.1;
                Ada.Text_Io.Put("A");
                delay 0.1;

                --|
                --| simulate complex calculations
                --|
                delay 1.0;
             end Semaphore_Seize;
          end loop;
          Ada.Text_Io.Put_Line(" task 1 Terminated");
          exception
          when Info : others => null;
          Ada.Text_Io.Put_Line(" Example_Of_Task_1_Task_Type exception raised: "
                               & Ada.Exceptions.Exception_Information(Info));
       end Independent_Process;
    end loop;
 end Example_Of_Task_1_Task_Type;
 --|
 --|    |==================|
 --|    | Example_of_task_2|
 --|    |==================|
 --|
 task body Example_Of_Task_2_Task_Type
    is
    --|
    --|  |==============================|
    --|  | M A I N     Example_of_task  |
    --|  |==============================|
    --|
    begin
    loop
       select
          accept Start
             do
             --|
             --|----------------------------------------|
             --| this is where the code of the task is  |
             --| executed in exclusion to anything else |
             --|----------------------------------------|
             --|
             Ada.Text_Io.Put_Line(" task 2 started");
          end Start;
          or
          terminate;
       end select;
       Independent_Process_Declaration :
       declare


       begin
          --|
          --|----------------------------------------|
          --| this is where the code of the task is  |
          --| executed AS AN INDEPENDENT PROCESS     |
          --|----------------------------------------|
          --|
          loop
             exit when Exit_The_Task;
             Semaphore_Seize :
             declare
             S : aliased Semaph.Semaphore_Type;
             Z : Semaph.Protects.Semaphore_Protect_Single(Semaphore => S'Access);
             begin
                --|
                --| put number on screen
                --|
                Ada.Text_Io.Put("B");
                delay 0.1;
                Ada.Text_Io.Put("B");
                delay 0.1;
                Ada.Text_Io.Put("B");
                delay 0.1;

                --|
                --| simulate complex calculations
                --|
                delay 0.95;
             end Semaphore_Seize;
          end loop;
          Ada.Text_Io.Put_Line(" task 2 Terminated");

          exception
          when Info : others => null;
          Ada.Text_Io.Put_Line("Example_Of_Task_2_Task_Type exception raised: "
                               & Ada.Exceptions.Exception_Information(Info));
       end Independent_Process_Declaration;
    end loop;
 end Example_Of_Task_2_Task_Type;
end Task_3_A;
```

10.10.7 fichier task_3.adb

```
--|           T A S K I N G     E X A M P L E
--|
```

```
--|
--
-- University of Paris 12 ADA Introductory Course Jan 2001
-- Daniel.Gaudry@wanadoo.fr
--
--| Version 1.00 october  12th 2001 initial coding

-- this very simple tasking example will put on the screen
-- AAA from one task and BBB from the other task
-- screen
-- the following difficulties will be illustrated:
--  the pattern ( normally AAABBBAAABBB.....)
-- will not be respencted if the semaphoer is not used
--  because the screen can receive characters from the two tasks
-- at the same time resulting in an incorect order
--  a semaphore will be illustrated to remedy this problem

with Ada.Text_Io;-- this is a routine already available in ada for i/o of  t e x t (strings)
with Task_3_A;
with Ada.Task_Identification;
--|                          |_______|
--|                          | TASK_3 |
--|                          |_______|
procedure Task_3
   is
   procedure Run_Tasks(Choice : in     Boolean)
      is
      begin
      --|
      --| reset exit flag
      --|
      Task_3_A.Exit_The_Task := False;
      --|
      --|start task number 1
      --|
      Task_3_A.Example_Of_Task_1_Task.Start;
      --|
      --|start task number 2
      --|
      Task_3_A.Example_Of_Task_2_Task.Start;
      --|
      --|Simulate something going on here for 30 seconds
      --|
      for I in 1 .. 30 loop
         delay 1.0;
         Ada.Text_Io.Put(' ');
      end loop;
      --|
      --| set exit flag => both tasks will exit
      --|
      Task_3_A.Exit_The_Task := True;
      delay 3.0;-- wait for tasks to terminate
   end Run_Tasks;



   begin
   Ada.Text_Io.Put_Line("the A's are from task 1, the B's from task 2 and the spaces from the main");
   --|
   --| put header and start both tasks without semaphore
   --|

   Run_Tasks(Choice => False);
   Ada.Text_Io.New_Line;
   --|
   --|is task number 1 still alive and running if so kill the task
   --|
   if Ada.Task_Identification.Is_Callable(T => Task_3_A.Example_Of_Task_1_Task'Identity)
      then
      abort Task_3_A.Example_Of_Task_1_Task;
   end if;
   --|
   --|is task number 2 still alive and running if so kill the task
   --|
   if Ada.Task_Identification.Is_Callable(T => Task_3_A.Example_Of_Task_2_Task'Identity)
      then
      abort Task_3_A.Example_Of_Task_2_Task;
   end if;

end Task_3;
```

10.11 QUATRIÈME EXEMPLE

10.11.1 Fichier a.adb

```
with Call_Task;

procedure A
   is
```

```
   begin
   Call_Task.Calcul(Number      => 70,
                    Task_Number => 1);
   Call_Task.Calcul(Number      => 60,
                    Task_Number => 2);
   Call_Task.Calcul(Number      => 50,
                    Task_Number => 3);
   Call_Task.Calcul(Number      => 40,
                    Task_Number => 4);
   Call_Task.Calcul(Number      => 30,
                    Task_Number => 5);
   Call_Task.Calcul(Number      => 20,
                    Task_Number => 6);
   Call_Task.Calcul(Number      => 10,
                    Task_Number => 7);
end A;
```

10.11.2 Fichier call_task.ads

```
package Call_Task
   is

   procedure Calcul(Number      : in     Positive;
                    Task_Number : in     Positive);

end Call_Task;
```

10.11.3 Fichier call_task.adb

```
with Ada.Text_Io;
with Task_Global;

package body Call_Task
   is
   procedure Calcul(Number      : in     Positive;
                    Task_Number : in     Positive)
      is
      Create_Task_Access : Task_Global.Create_Task_Access_Type := null;
      Local_Data_Access  : Task_Global.Data_Access_Type        := null;
      Data               : Task_Global.Data_Type               := (N => Number);
      begin
      --|
      --|create a fresh new data set and initialiaze it
      --|
      Local_Data_Access := new Task_Global.Data_Type'(Data);
      --|
      --|create a fresh new task with D I S C R I M I N A N T   I N I T I A L I Z E D      {to pass the data}
      --|
      Create_Task_Access := new Task_Global.Create_Task_Type(Data_Access => Local_Data_Access);
      --|
      --|start the task
      --|
      select
         Create_Task_Access.Create(Task_Number => Task_Number);
         or
         delay 0.2;
         --|
         --|The task wasn't ready, skip the command.
         --|
         Ada.Text_Io.Put_Line("task not ready");
      end select;
   end Calcul;
end Call_Task;
```

10.11.4 Fichier task_global.ads

```
Package Task_Global
   is
   type Data_Type is record
      N : Integer;
   end record;
   --|
   --| global data types
   --|
   type Data_Access_Type is access all Data_Type;
   --|
   --|task create
   --|
   task type Create_Task_Type(Data_Access : Data_Access_Type)
      is
      pragma Storage_Size(12 * 1024);

      entry Create(Task_Number : in     Integer);
   end Create_Task_Type;
   --|
   --|define access type for the task so that you can craeate several
   --|
   type Create_Task_Access_Type is access Create_Task_Type;
end Task_Global;
```

10.11.5 Fichier task_global.adb

```
with Ada.Text_Io;


package body Task_Global
   is
   task body Create_Task_Type
      is
      Dummy : Integer;
      begin
      loop
         select
            accept Create(Task_Number : in     Integer)
               do
               Dummy := Task_Number;
            end Create;
            or
            terminate;
         end select;

         Calcul_Of_Factorial :
         declare
         X      : Data_Access_Type := Data_Access;
         Result : Long_Float       := 1.0;
         begin
            Ada.Text_Io.Put_Line("task number "
                                 & Integer'Image(Dummy));
            for I in Long_Integer(2) .. Long_Integer(X.all.N) loop
               Result := Result * Long_Float(I);
               delay 0.01;
            end loop;
            Ada.Text_Io.Put_Line("factorial"
                                 & Integer'Image(X.all.N)
                                 & " ="
                                 & Long_Float'Image(Result));
         end Calcul_Of_Factorial;
      end loop;
   end Create_Task_Type;
end Task_Global;
```

10.12 CINQUIÈME EXEMPLE

10.12.1 Fichier Philosopher_Main.adb

```
with Text_Io;
with Philosopher_Task;

procedure Philosopher_Main
   is
   begin
   --| demarrer chaque tache une pour chaque invité
   Philosopher_Task.Philosophes(1).Get_Id(1,"Gerry");
   Philosopher_Task.Philosophes(2).Get_Id(2,"Randy");
   Philosopher_Task.Philosophes(3).Get_Id(3,"Dan  ");
   Philosopher_Task.Philosophes(4).Get_Id(4,"Carl ");
   Philosopher_Task.Philosophes(5).Get_Id(5,"Jean ");

end Philosopher_Main;
```

10.12.2 Fichier Philosopher_Data.ads

```
-- This example, and some code was
-- borrowed from the Ada teaching book "Ada: An Advanced Introduction", by
-- Narian Gehani, Prentice-Hall, 1983, pp 177-182.
--
-------------------------------------------------------------------------------
--===========================================================================
--EDIT HISTORY:
```

```
-- Created "The Mortal Dining Philosophers" 11/13/87 taK
-- Converted to "The Five Dining Ada Gurus" 11/19/87 taK
-- Prettied up for outside use              12/17/87 RLB
-- put in french D.G 02/26/2007
------------------------------------------------------------------------------

package Philosopher_Data
   is


   Num_Philosophes : constant := 5;

   subtype Id is Integer range 1 .. Num_Philosophes;

   subtype Philosophe_Name is String(1 .. 5);


end Philosopher_Data;
```

10.12.3 Fichier Philosopher_Task.ads

```
with Philosopher_Data;

package Philosopher_Task
   is

   task type Ada_Philosophe
      is
      entry Get_Id(J    : in     Philosopher_Data.Id;
                   Name : in     Philosopher_Data.Philosophe_Name);
   end Ada_Philosophe;

   task type Fork
      is
      entry Pick_Up;
      entry Put_Down;
   end Fork;

   task Host
   is
   entry Enter(Name : Philosopher_Data.Philosophe_Name);
   entry Leave(Name : Philosopher_Data.Philosophe_Name);
   end Host;
   --|declaration des taches :
   --|Il y a 5 fourchettes
   Forks : array (Philosopher_Data.Id) of Fork;
   --|il y a 5 philosophes
   Philosophes : array (Philosopher_Data.Id) of Ada_Philosophe;

end Philosopher_Task;
```

10.12.4 Fichier Philosopher_Task.adb

```
with Random_Normal;
with Ada.Text_Io;

package body Philosopher_Task
  is
  --|
  --| cette tache a pour but d'obliger une fourchette  PRECISE
  --| ( il y a 5 fourchettes) a etre reposee sur la
  --| table avant de pourvoir etre prise par une autre personne. cette tache se termine
  --| quant Ada_PHILOSOPHE se termine
  --|
  task body Fork
     is
     begin
     --|
     --| boucle classique des taches en ada
     --|
     loop
        select
           --|
           --|  entree prendre une fourchette      P R E C I S E
           --|                                     =============
           accept Pick_Up
              do
              Ada.Text_Io.Put_Line("    fourchette prise");
              null;
           end Pick_Up;
           --|
           --|  entree reposer une fourchette      P R E C I S E
           --|                                     =============
```

```
         accept Put_Down
            do
            Ada.Text_Io.Put_Line("    fourchette posee");
            null;
         end Put_Down;
         --|
         --|  suite syntaxe tache
         --|
         or
         terminate;
      end select;
   end loop;
end Fork;
--|
--|     GESTION ENTREE SORTIE DES PHILOSOPHES
--|
task body Host
   is
   --|
   --| global pour host
   --|
   I : Integer := 0;--number of philosophers in the room
   begin
   loop
      select
         --|
         --| test si entree possible
         --|
         when I < 4 =>
         --a philosopher can enter if there are less then 4
         --philosophers in the dining room.
         --|
         --| ok
         --|
         accept Enter(Name : Philosopher_Data.Philosophe_Name)
            do
            Ada.Text_Io.Put_Line("Bonjour "
                                 & Name);
            null;
         end Enter;
         --|
         --| nombre de philosophes
         --|
         I := I + 1;
         --|
         --|
         --|
         or
         --|
         --|
         --|
         accept Leave(Name : Philosopher_Data.Philosophe_Name)
            do
            Ada.Text_Io.Put_Line("AU revoir "
                                 & Name);
            null;
         end Leave;
         --|
         --|nombre de philosophes
         --|
         I := I - 1;

         or
         terminate;
      end select;
   end loop;
end Host;
--|
--|  GESTION DU RESTAURANT ET DE LA TABLE
--|
task body Ada_Philosophe
   is
   --|
   --|
   --|
   I               : Philosopher_Data.Id;--numero du philosophe
   Max_Eats_Per_Day  : constant := 4;
   Times_Eaten       : Integer  := 0;
   Left_Fork_Number  : Philosopher_Data.Id;
   Right_Fork_Number : Philosopher_Data.Id;
   My_Name           : Philosopher_Data.Philosophe_Name;
   Random_Delay      : Long_Float;
   begin
   --|
   --| noter la syntaxe differente de cette tache
   --|
   accept Get_Id(J    : in     Philosopher_Data.Id;
                 Name : in     Philosopher_Data.Philosophe_Name)
      do
      -- Get the identification number
      --|
      --|
      --|
      I       := J;-- j   disparait aprs end Get_Id, il est mis en global pour la suite
      --|
      --|
      --|
```

```
      Ada.Text_Io.Put_Line(My_Name
                           & " marche vers le restaurant");
      --|
      --| cette partie du code est en arret jusqu'a la fin du delai.
      --|
      Random_Delay := Random_Normal.Draw_Random(Mean               => 2.0,
                                                Standard_Deviation => 1.0,
                                                Max                => Long_Float'Last,
                                                Min                => 0.0);
      Ada.Text_Io.Put_Line(My_Name
                           & " attend pendant "
                           & Long_Float'Image(Random_Delay)
                           & " en marchant");
      --|
      --| attente
      --|
      delay Duration(Random_Delay);
   end Get_Id;
   --|
   --| les fourchettes sont numerotees a partir du numero du
   --| philosophe (i):
   --|
   --|
   --|
   --|           PHILOSOPHE         FOURCHETTE GAUCHE       FOURCHETTE DROITE
   --|
   --|              1                     1                       2
   --|              2                     2                       3
   --|              3                     3                       4
   --|              4                     4                       5
   --|              5                     5                       1
   --|
   --|
   Left_Fork_Number  := I;
   Right_Fork_Number := I mod 5 + 1;
   --|
   --|  le philosophe peut manger 4 fois
   --|
   while Times_Eaten /= Max_Eats_Per_Day loop

      Ada.Text_Io.Put_Line(My_Name
                           & "discute avec son voisin ");
      --|
      --| calcul de l'attente avant d'entrer dans le restaurant
      --|
      Random_Delay := Random_Normal.Draw_Random(Mean               => 20.0,
                                                Standard_Deviation => 3.0,
                                                Max                => Long_Float'Last,
                                                Min                => 0.0);

      Ada.Text_Io.Put_Line(My_Name
                           & " lis le menu"
                           & Long_Float'Image(Random_Delay));
      --|
      --|attente avant d'entrer dans le restaurant
      --|
      delay Duration(Random_Delay);
      --|
      --| appel a une entree de tache: doit attendre dans la queue correspondante
      --| si elle n'est pas vide
      --|
      Host.Enter(My_Name);

      Ada.Text_Io.Put_Line(My_Name
                           & " entre et prend un siege ");
      --|
      --| appel a une entree de tache: doit attendre dans la queue correspondante
      --| si elle n'est pas vide
      --|
      Forks(Right_Fork_Number).Pick_Up;
      Ada.Text_Io.Put_Line(My_Name
                           & " prend sa fourchette droite ");
      --|
      --| appel a une entree de tache: doit attendre dans la queue correspondante
      --| si elle n'est pas vide
      --|
      Forks(Left_Fork_Number).Pick_Up;
      Ada.Text_Io.Put_Line(My_Name
                           & " prend sa fourchette droite ");
      --|
      --|calcul du temps pour manger
      --|
      Random_Delay := Random_Normal.Draw_Random(Mean               => 8.0,
                                                Standard_Deviation => 2.0,
                                                Max                => Long_Float'Last,
                                                Min                => 0.0);

      Ada.Text_Io.Put_Line(My_Name
                           & " va manger pendant "
                           & Long_Float'Image(Random_Delay));
      --|
      --|attente du temps pour manger
      --|
      delay Duration(Random_Delay);
      --|
      --| repas termine: poser la fourchette gauche : elle devient libre
      --| appel a une entree de tache: doit attendre dans la queue correspondante
```

```
        --| si elle n'est pas vide
        --|
        Forks(Left_Fork_Number).Put_Down;
        Ada.Text_Io.Put_Line(My_Name
                             & " a fini et repose sa fourchette gauche");
        --|
        --| repas termine: poser la fourchette droite : elle devient libre
        --|appel a une entree de tache: doit attendre dans la queue correspondante
        --| si elle n'est pas vide
        --|
        Forks(Right_Fork_Number).Put_Down;
        Ada.Text_Io.Put_Line(My_Name
                             & " repose sa fourchette droite");
        --|
        --| nombre de repas
        --|
        Times_Eaten := Times_Eaten + 1;

        Ada.Text_Io.Put_Line(My_Name
                             & " se leve et part");
        --|
        --|appel a une entree de tache: doit attendre dans la queue correspondante
        --| si elle n'est pas vide
        --|
        Host.Leave(My_Name);
     end loop;

     Ada.Text_Io.Put_Line(My_Name
                          & " va se coucher");
     -- c'est la fin de cette tache elle disparait
  end Ada_Philosophe;

end Philosopher_Task;
```

10.12.5 Fichier Random_Normal.ads

```
package Random_Normal
   is

   function Draw_Random(Mean               : in     Long_Float;
                        Standard_Deviation : in     Long_Float;
                        Max                : in     Long_Float := Long_Float'Last;
                        Min                : in     Long_Float := Long_Float'First)
   return Long_Float;

   private

   subtype Index_Type is Integer range 0 .. 623;
   type Mod_32_Type is mod 2 ** 32;
   type Mod_64_Type is mod 2 ** 64;
   type Mod_32_Array_Type is array (Index_Type'Range) of Mod_32_Type;

   Mod_32_Array : Mod_32_Array_Type := (others => Mod_32_Type'First);
   Index        : Index_Type        := Index_Type'First;
   First_Time   : Boolean           := True;

end Random_Normal;
```

10.12.6 Fichier Random_Normal.adb

```
--                    M E R S E N N E   T W I S T E R
--                    ==============================
-- Pseudocode
-- The following generates uniformly 32 bit integers in the range [0, 2^32 - 1]
-- with the MT19937 algorithm:
--  // Create a length 624 array to store the state of the generator
--  var int[0..623] MT
--  var int y
--  // Initialise the generator from a seed
--  function initialiseGenerator ( 32-bit int seed ) {
--      MT[0] := seed
--      for i from 1 to 623 { // loop over each other element
--          MT[i] := last_32bits_of((69069 * MT[i-1]) + 1)
--      }
--  }
--
--  // Generate an array of 624 untempered numbers
--  function generateNumbers() {
--      for i from 0 to 622 {
--          y := 32nd_bit_of(MT[i]) + last_31bits_of(MT[i+1])
--          if y even {
```

```
--             MT[i] := MT[(i + 397) % 624] bitwise_xor (right_shift_by_1_bit(y))
--         } else if y odd {
--             MT[i] := MT[(i + 397) % 624] bitwise_xor (right_shift_by_1_bit(y))
--             bitwise_xor (2567483615)
--         }
--     }
--     y := 32nd_bit_of(MT[623]) + last_31bits_of(MT[0])
--     if y even {
--         MT[623] := MT[396] bitwise_xor (right_shift_by_1_bit(y))
--     } else if y odd {
--         MT[623] := MT[396] bitwise_xor (right_shift_by_1_bit(y))
--         bitwise_xor (2567483615)
--     }
--  }
--
--  // Extract a tempered pseudorandom number based on the i-th value
--  function extractNumber(int i) {
--     y := MT[i]
--     y := y bitwise_xor (right_shift_by_11_bits(y))
--     y := y bitwise_xor (left_shift_by_7_bits(y) bitwise_and (2636928640))
--     y := y bitwise_xor (left_shift_by_15_bits(y) bitwise_and (4022730752))
--     y := y bitwise_xor (right_shift_by_18_bits(y))
--     return y
--  }
--
--
-- Reference
-- M. Matsumoto and T. Nishimura, Mersenne twister: A 623-dimensionally
-- equidistributed uniform pseudorandom number generator,
-- ACM Trans. on Modeling and Computer Simulations, 1998.



with Ada.Exceptions;
with Ada.Numerics.Generic_Elementary_Functions;-- for **, sin, log, exp   ......
with Ada.Text_Io;

package body Random_Normal
   is
   --|
   --|             |                               |
   --|             |    Initialize_Generator       |
   --|             |_______________________________|
   --|

   --|
   --| Initialise the generator from a seed
   --|
   procedure Initialize_Generator(Seed : in     Mod_32_Type := 5489)
      is
      --|
      --| needed to avoid 32 bits to overflow when *1812433253 ( see below)
      --|
      Mod_64 : Mod_64_Type;
      begin
      --|
      --| get start from seed
      --|
      Mod_32_Array(0) := Seed;--;and 16#FFFFFFFF#;
      --|
      --| loop to fill the array from seed ( done once at init)
      --|
      for I in 1 .. Mod_32_Array'Last loop
         --|
         --|no risk to overflow: changed to 64 bits storage before operation
         --|
         Mod_64 := Mod_64_Type(Mod_32_Array(I - 1)
                               xor Mod_32_Array(I - 1) / 2 ** 30) + Mod_64_Type(I);
         --|
         --|  here the risk is avoided 32 bits *1812433253 cannot overflow in 64 bit storage
         --|
         Mod_64 := 1812433253 * Mod_64;
         --|
         --| zero all high order bits keeping only 32 and back to 32 bit storage
         --|
         Mod_32_Array(I) := Mod_32_Type(Mod_64 and 16#FFFFFFFF#);
      end loop;
      --|
      --|                                        ========================
      --|                                        e r r o r   h a n d l e r
      --|                                        ========================
      --|
      exception

      when Programing_Error : others =>
      Ada.Text_Io.Put("Random_Normal.Initialize_Generator"
                      & Ada.Exceptions.Exception_Information(Programing_Error));
   end Initialize_Generator;
   --|
   --|             |                               |
   --|             | Generate_a_new_array          |
   --|             |_______________________________|
   --|
   procedure Generate_A_New_Array
      is
      -- Generate a new array of 624 untempered numbers from the previous state of the array
```

```
      Y : Mod_32_Type;
      begin
      --|
      --|loop
      --|
      for I in Index_Type'First .. (Index_Type'Last - 1) loop
         Y := (Mod_32_Array(I) and 16#80000000#) + (Mod_32_Array(I + 1) and 16#7FFFFFFF#);
         --|
         --| odd or even
         --|
         if (Y and 1) = 1
            then
            Mod_32_Array(I) := Mod_32_Array((I + 397) mod (Index_Type'Last + 1))
            xor Y / 2 xor 2567483615;
            else
            Mod_32_Array(I) := Mod_32_Array((I + 397) mod (Index_Type'Last + 1)) xor Y / 2;
         end if;
      end loop;
      --|
      --| last one
      --|
      Y := (Mod_32_Array(Index_Type'Last) and 16#80000000#) + (Mod_32_Array(0) and 16#7FFFFFFF#);
      --|
      --|odd or even
      --|
      if (Y and 1) = 1
         then
         Mod_32_Array(Index_Type'Last) := Mod_32_Array(Index_Type'Last) xor Y / 2 xor 2567483615;
         else
         Mod_32_Array(Index_Type'Last) := Mod_32_Array(Index_Type'Last) xor Y / 2;
      end if;
      --|
      --|                                    ========================
      --|                                    e r r o r   h a n d l e r
      --|                                    ========================
      --|
      exception

      when Programing_Error : others =>
      Ada.Text_Io.Put("Random_Normal.Generate_A_New_Array"
                      & Ada.Exceptions.Exception_Information(Programing_Error));
   end Generate_A_New_Array;
   --|
   --|              |                    |
   --|              | Generate_Numbers   |
   --|              |____________________|
   --|
   function Generate_Numbers
      return Mod_32_Type
      is
      Y : Mod_32_Type;
      begin
      --|
      --| init
      --|
      if First_Time
         then
         Initialize_Generator(Seed => 5489);
         First_Time := False;
         Index      := 0;
         Generate_A_New_Array;
      end if;
      --|
      --| see algo
      --|
      Y := Mod_32_Array(Index);
      Y := Y xor (Y / 2 ** 11);
      Y := Y xor (Y * 2 ** 7 and 2636928640);
      Y := Y xor (Y * 2 ** 15 and 4022730752);
      Y := Y xor (Y / 2 ** 18);
      --|
      --|all used up ? if so shuffle array, if not increment index for next call
      --|
      if Index = Index_Type'Last
         then
         Index := Index_Type'First;
         Generate_A_New_Array;
         else
         Index := 1 + Index;
      end if;
      --|
      --|debug
      --|
      --Ada.Text_Io.Put(" Index"
      --& Index_Type'Image(Index)
      --& " Y"
      --& Mod_32_Type'Image(Y));
      --|
      --|done
      --|
      return Y;
      --|
      --|                                    ========================
      --|                                    e r r o r   h a n d l e r
      --|                                    ========================
      --|
      exception
```

```
   when Programing_Error : others =>
   Ada.Text_Io.Put("Random_Normal.Generate_Numbers"
                   & Ada.Exceptions.Exception_Information(Programing_Error));
   return 0;
end Generate_Numbers;
--|
--|              |                  |
--|              | Draw_Random      |
--|              |__________________|
--|
function Draw_Random(Mean               : in     Long_Float;
                     Standard_Deviation : in     Long_Float;
                     Max                : in     Long_Float := Long_Float'Last;
                     Min                : in     Long_Float := Long_Float'First)
   return Long_Float
   is
   --|
   --|              |                  |
   --|              |     Get_One      |
   --|              |__________________|
   --|
   function Get_One
      return Long_Float
      is
      --|
      --| needed to get log and ** on float
      --|
      package Math is new Ada.Numerics.Generic_Elementary_Functions(Float_Type => Long_Float);

      function Ln(X : in     Long_Float)
         return Long_Float
      renames Math.Log;

      function "**" (X : in Long_Float;
                     Y : in     Long_Float)
         return Long_Float
      renames Math."**";

      X             : Long_Float;
      Y             : Long_Float;
      Radius        : Long_Float;
      Normal_Random : Long_Float;

      begin
      --|
      --|box muller transform ( from a random 0..1 to normal distribution)
      --| z= x square root(-2Ln r/r)
      --|
      loop
         --|
         --| get two random in the range 0 .. 2**32
         --| and change range from 0 .. 2**32 to 0.0 .. 1.0
         --|
         Y := Long_Float(Generate_Numbers) / (2.0 ** 32 - 1.0);
         X := Long_Float(Generate_Numbers) / (2.0 ** 32 - 1.0);
         --|
         --|change to -1.0 .. 1.0
         --|
         X := X * 2.0 - 1.0;
         Y := Y * 2.0 - 1.0;
         --|
         --|check if within unit circle
         --|
         Radius := X ** 2 + Y ** 2;

         exit when Radius > 0.0 and Radius < 1.0;
      end loop;
      --|
      --| change into normal distribution with mean 0 and SD=1
      --|
      Normal_Random := X * (- 2.0 * Ln(Radius) / Radius) ** 0.5;
      --|
      --|change from normal with mean 0 and SD=1 into mean and sd as needed
      --|
      Normal_Random := Normal_Random * Standard_Deviation + Mean;
      --|
      --| done
      --|
      return Normal_Random;
      --|
      --|                                    ========================
      --|                                    e r r o r   h a n d l e r
      --|                                    ========================
      --|
      exception

      when Programing_Error : others =>
      Ada.Text_Io.Put("Random_Normal.Get_One"
                      & Ada.Exceptions.Exception_Information(Programing_Error));
      return 0.0;
   end Get_One;
   --|              | Draw_Random      |
   Result : Long_Float;
   begin
   --|
   --| check if within limits, if not try again
```

```
      --|
      loop
         Result := Get_One;
         --Ada.Text_Io.Put(  " Result"
         --& Long_Float'Image(Result));
         exit when Result in Min .. Max;
      end loop;
      --|
      --|done
      --|
      return Result;
      --|
      --|                                    ========================
      --|                                    e r r o r   h a n d l e r
      --|                                    ========================
      --|
      exception

      when Programing_Error : others =>
      Ada.Text_Io.Put("Random_Normal.Draw_Random"
                      & Ada.Exceptions.Exception_Information(Programing_Error));
      return 0.0;
   end Draw_Random;
end Random_Normal;
```

11 ALGORITHMIQUE

11.1 INTRODUCTION

Les structures de base permettent une approche systématique du ou des problèmes posés car elles sont spécialisés de par leur propriétés.

11.1.1 Grammaires et langage formel, automates à états finis

Ils permettent de valider une entrée utilisateur, adresse, expression,.. de faire le découpage de données en éléments individuels, de reconnaissance de mots d'un langage.

11.1.2 Table de hashing

Une table de hashing permet la recherche, l'ajout et la suppression d'enregistrements en optimisant la rapidité des opérations.

11.1.3 Stack

Un stack permet de stocker des informations et de les lire dans l'ordre inverse de l'ajout.

11.1.4 Queue

Une queue permet de stocker des informations et de les lire dans l'ordre de l'ajout.

11.1.5 Arbre binaire

Un arbre binaire permet le stockage de données en ordonnant leur valeurs. Il peut être lu (entres autre) de façon à ce que les données soit triées en ordre croissant. Il permet un accès aux données en un temps au plus égal à O(LogN). Il permet toutes sorte d'applications.

11.1.6 Arbre dictionnaire

Un arbre dictionnaire permet le stockage de mots et leur recherche en un temps de l'ordre de O(taille du mot).

11.1.7 Graphe

Un graphe permet l'association de données en créant des "chemins" entre elles. Il peut être parcouru de diverses manières y compris par la recherche du plus court chemin entre deux données.

11.1.8 Tris

Permet d'ordonner des données selon un opérateur de comparaison défini par l'utilisateur.

11.1.9 Ensemble

Permet toutes les opérations classiques sur les ensembles: union, intersection,.....

11.1.10 Anneau

C'est une liste circulaire double.

11.2 GRAMMAIRES ET LANGAGE FORMEL, AUTOMATES À ÉTATS FINIS

11.2.1 Les abréviations

Les abréviations utilisées sont les suivantes :

{ xxxxxx **}**	veut dire	**0 à n répétition**s de xxxxxx
[yyyyyyy **]**	veut dire	yyyyyy est **optionnel**
|	veut dire	**ou**
::=	veut dire	**défini par**
<AA>	AA non terminal (utilisé pour définir la grammaire)	
'BB'	BB terminal (caractère, ou mot appartenant au language)	

11.2.2 Fonction

Une fonction sera définie comme suit:
'Fonction' <nom_de_fonction>(<nom_de_variable> : <type_de_variable>
{';' <nom_de_variable> : <type_de_variable>**}**)
'renvoie' <type_de_variable>
<déclaration> **'Début'** <instruction> **{**<instruction>**}]**
'fin fonction'

11.2.3 Procédure

Une procédure sera définie comme suit:
'Procédure' <nom_de procédure> (<nom_de_variable> : <type_de_variable>
{';' <nom_de_variable> : <type_de_variable> **}**)
<déclaration> **'Début'** <instruction> **{**<instruction>**}]**
'fin procédure'

11.2.4 L'algorithme

algorithme ::= <nom_algo>
<déclaration>
<block_instructions>

11.2.5 La déclaration

déclaration ::= **'Variable'**
{<nom_de_variable> **':'** <type_de_variable>**}**

11.2.6 L'instruction

instruction	::=	<nom_de_variable > '←' <expression>
	|	<expression_multiple>
	|	'renvoi' '('<nom_de_variable > ')'

11.2.7 L'expression multiple

expression_multiple	::=	<test>
	|	<itération>
	|	<appel_procedure>

11.2.8 Le test

test	::=	'si' <expression >
		'alors'
		<instruction>
		{<instruction>}
		['sinon'
		<instruction>
		{<instruction>}]
		'fin si'

11.2.9 L'itération

itération	::=	'tant que'
		<expression>
		'boucle'
		<instruction>
		{<instruction>}
		'fin boucle'

11.2.10 Le block d'instruction

block_instructions	::=	'Début'
		<instruction>
		{<instruction>}]
		'Fin'

11.3 INVARIANT DE BOUCLE ET VALIDATION

Un invariant de boucle est une relation entre les variables d'un programme possédant les propriétés suivantes:

- Cette relation est vraie avant l'entrée dans la boucle.
- Cette relation est vraie avant chaque itération (tour de la boucle) et reste vraie avant la dernière itération de la boucle.
- Quant la boucle est terminée, l'invariant de boucle possède une propriété qui permet de démontrer la validité de l'algorithme par comparaison avec une propriété mathématique du problème à résoudre.

11.3.1 Utilisation de l'invariant de boucle

Comme premier exemple d'utilisation, nous considéreront un programme qui fait la somme de tous les éléments contenus dans un tableau. Soit T le tableau. La somme S de tous les éléments du tableau se définit comme:
Sum = T(1)+.. + T(i). Cette formulation souligne le fait que l'invariant est une expression qui est toujours VRAIE

Voici le code d'une telle fonction:

```
function Sum(T : in     My_Array_Type)
 return Integer
 is
 S : Integer;
 begin
 S := 0;
 for I in T'Range loop
 -- s <= T(1) + ... + T(i-1)
    S := S + T(I);
 end loop;
-- s <= T(1) + ... + T(N)
 return S;
end Sum;
```

Les commentaires indiquent l'usage de l'invariant de boucle pour justifier de la validité du code. La première fois que l'itération est exécutée, i=1 et s=0 avant et s=t(1) après la première itération. L'utilisation d'une boucle « for » cache une partie du fonctionnement. Un programme avec une boucle « while » est beaucoup plus clair:

```
function Sum(T : in     My_Array_Type)
  return Integer
  is
  S : Integer;
  I : Integer := T'First;
  begin
  S := 0;
  while I in T'Range loop
  -- s <= T(1) + ... + T(i-1)
     S := S + T(I);
     I := 1 + I;
  end loop;
-- s <= T(1) + ... + T(n)
  return S;
end Sum;
```

Maintenant, nous pouvons écrire les conditions de validité d'une itération :

- L'invariant est vrai avant le début de la boucle : S := 0;
- L'invariant est vrai pendant l'exécution de la boucle.
- L'indice indique l'avancement du calcul.
- La boucle se termine avec la dernière itération.
- Après l'exécution l'invariant est toujours vrai et correspond au but recherché.

11.4 COMPLEXITÉ DES ALGORITHMES

11.5 TEMPS D'EXÉCUTION DES ALGORITHMES

12 LES STRUCTURES CLASSIQUES

Une structure est utilisée pour enregistrer, stocker et traiter des données selon des propriétés qui dépendent du type de structure. Pour simplifier et rationaliser, toutes les structures, seront munies d'opérations utilisant les mêmes noms (y compris dans le cas ou des noms spécifiques ou consacrés par l'usage existent déjà). Ces structures auront leur intégrité protégée: les seuls accès possibles aux données et à leur stockage se feront obligatoirement en utilisant les opérations dont la structure est munie.

12.1 L' ITÉRATEUR

Dans toutes ces structures, l'accès direct au stockage utilisé sera impossible, il faut donc proposer à l'utilisateur toutes les opérations nécessaires. L'une d'entre elle, l'accès direct en utilisant, par exemple, l'indice du tableau ou sont stockées les données devient impossible. Une opération spéciale remplacera cet accès direct:
Toutes ces structures seront donc munies d'une opération spéciale: un **itérateur**. Cette abstraction, qui remplace une boucle accédant directement au stockage utilisé par la structure, est un objet qui permet de visiter tous les éléments stockés dans la structure. La visite se fera dans l'ordre spécial déterminé par le type de la structure utilisée (y compris sans aucun ordre particulier si la structure utilisée n'en spécifie aucun). Cet itérateur sera divisé en plusieurs sous-opérations qui fonctionnent comme suit pour que la variable **élément** prenne successivement toutes les valeurs des éléments stockées dans la structure:

Pseudo-code d'utilisation de l'itérateur :

un_itérateur ⟵ **initialisation**

boucle tant que non **fini(**un_itérateur**)**

élément ⟵ **valeur(**un_itérateur**)**

suivant(un_itérateur**)**

fin boucle

Toutes les structures seront munis des opérations suivantes permettant d'utiliser l'itérateur:

initialisation

Met à jour toutes les variables de l' itérateur, charge la première valeur et renvoie une variable de type itérateur (on peut avoir besoin de plus d'un itérateur).

fini

Renvoie faux tant qu'il reste des éléments non visités, vrai quant tous les éléments sont visités

valeur

Renvoie l'élément courant.

suivant

Passe à l'élément suivant si il existe.

12.2 INTRODUCTION

Les structures classiques seront codées en utilisant le concept de générique. De cette façon, elles pourront être utilisées quelque soit le type de variable.

12.3 LA TABLE DE HASHING

Une table de hashing est une structure permettant le stocker et de retrouver **très** rapidement des données. L'ajout, la recherche et l'effacement sont basées sur une clé pour laquelle on peut définir l'opérateur logique égal. L'algorithme de la version à taille fixe nécessite également de définir deux éléments spéciaux: **vide** et **déjà utilisé**. L'ordre des données à l'intérieur de la structure peut être considéré comme aléatoire. L'itérateur visitera donc chaque élément mais sans aucun ordre.

L'idée de base de la table de hashing consiste à transformer la clé en l'indice d'un tableau en dispersant autant que possible les valeurs numériques obtenues pour améliorer la vitesse. Pour l'implémentation utilisant un tableau simple de taille fixe, Il faut utiliser un tableau dont la taille est un nombre premier plus grand d'un facteur 2 que le nombre prévu d'éléments à y ajouter.
Il est évident qu'il existe TOUJOURS plus d'éléments possibles que de places dans le tableau. Il est donc certain que deux clés différentes seront traduites dans le même indice. Cela s'appelle une collision. Son traitement est différent selon le type d'implémentation choisi (taille fixe dans un tableau simple ou taille variable dans un tableau chaîné avec pointeurs).

12.3.1 Vitesse de traitement

Soit α le rapport du nombre d'élément à la taille du tableau

Opération	Tableau Simple	Tableau Chaîné
Addition ou Recherche infructueuse	O (1/(1-α))	O (1+α)
Recherche ou addition fructueuse	O (1/α *Ln(1+α)	O (1+α)

Ces résultats ne sont réalistes que si le nombre d'éléments est petit, dans le cas contraire des contraintes indépendantes du raisonnement algorithmique entrent en jeu pendant l'exécution: saut de page de mémoire, défaut de page mémoire, passage en mémoire virtuelle avec accès disque,.....

12.3.2 Traduction de la clé en un nombre

C'est l'étape fondamentale de fonctionnement d'une table de hashing. Elle se décompose généralement en deux étapes:

1. Transformation des caractères de la clé en un nombre quelconque (optionnel si la clé est déjà numérique)
2. Transformation du nombre quelconque en indice du tableau compris entre le premier et le dernier.

Il existe de nombreuse façons de faire. En voici quelque unes:

12.3.2.1 Numérisation par addition

Les valeurs ASCII des caractères de la clé sont additionnées modulo N.

12.3.2.2 Partage d'emplacement mémoire

Un emplacement mémoire, généralement 32 bits, est partagé entre un entier sans signe codé sur 32 bits et une chaîne de 4 caractères codées selon le code ASCII. On découpe la série de caractères à coder 4 par 4 et on additionne les valeurs numériques correspondant à l'entier sans signe partageant le même espace mémoire (sans retenue dans ce cas). Si le nombre de caractères n'est pas divisible par 4 on complète avec des espaces ou toute autre valeur.
Exemple avec abcdefghij
Pour mieux comprendre prenons abcd en binaire sur 32 bits.
abcd en binaire sur 32 bits :01100100011000110110001001100001
Notez le fait que a est stocké à droite du mot sur 32 bits, en non pas à gauche comme on pourrait le penser : il y a deux type de stockage possibles appelés little endian et big endian et le constructeur du matériel a choisi l'un des deux modes, cela ne change pas la validité de ce code, mais ce n'est pas toujours le cas.

```
abcd    traduit en entier        1684234849      total  1684234849
efgh    traduit en entier        1751606885      total  1288358087      noter l'overflow
ββij    traduit en entier        1785274400      total   926148840
```

(avec β= espace)
la valeur numérisée de abcdefghij = 926148840

Le code ada:

```
with Unchecked_Conversion;
with Ada.Text_Io;

......

  Modulo : constant Integer := 2 ** 31 - 1;-- par exemple


  function Hash(String_To_Be_Hashed : in     String)
    return Integer
    is

    type Hash_Value_Type is mod Modulo;

    Flag  : Boolean := False;
    First : Integer;
    Last  : Integer;
    Temp  : Hash_Value_Type := Hash_Value_Type'First;
    Value : Hash_Value_Type := Hash_Value_Type'First;
    --|
    --| CALCULATE THE NUMBER OF CHARACTERS THAT FIT IN AN   INTEGER
    --|
    Chars_In_Int : constant := Integer'Size / Character'Size;
    --|
    --| CREATE ANd DECLARE SUCH A STRING
```

```
    --|
    subtype Substring is String(1 .. Chars_In_Int);
    S : Substring;
    --|
    --| THIS FUNCTION :
    --| ASSIGNS THE SAME ADDRESS TO THE STRING TYPE SUBSTRING  AND AN INTEGER AND
    --| RETURNS THE INTEGER VALUE
    --|
    function Substring_To_Integer is new Unchecked_Conversion(Source => Substring,
                                                              Target => Integer);
    begin
    --|
    --| GET THE FIRST CHARACTER TO HASH
    --|
    First := String_To_Be_Hashed'First;
    --|
    --| LOOP TILL ALL CHARACTERS HASHED
    --|
    Main_Cycle :
      loop
      --|
      --| SET THE END OF THE SLICE TO BE CUT
      --|
      Last := First + Chars_In_Int - 1;
      --|
      --| IS IT THE END OF THE STRING TO HASH
      --|
      if Last >= String_To_Be_Hashed'Last
        then
        --|
        --| YES PAD WITH SPACES
        --|
        S := (S'Range => ' ');

        S((Last - String_To_Be_Hashed'Last + 1) .. Chars_In_Int)
          := String_To_Be_Hashed(First .. String_To_Be_Hashed'Last);
        --|
        --| THIS IS THE END OF THE HASHING
        --|
        Flag := True;
        else
        --|
        --| NO CUT A SLICE
        --|
        S := String_To_Be_Hashed(First .. Last);
      end if;
      --|
      --| TRANSLATE STRING TO INTEGER AND GET MODULO VALUE
      --|
      Temp := Hash_Value_Type(Substring_To_Integer(S) mod Modulo);
      --|
      --| KEEP ADDING (MODULO)
      --|
      Value := Value + Temp;
      --|
      --| over exit
      --|
      exit Main_Cycle when Flag;
      --|
      --| not over go to next slice
      --|
      First := Last + 1;
    end loop Main_Cycle;
    --|
    --| return integer value
    --|
    return Integer(Value);
  end Hash;
```

12.3.2.3 Attribution d'une valeur numérique à chaque caractère

On affecte, généralement par tirage au hasard, un chiffre à chaque caractère et on fait la somme des valeurs correspondant aux caractères à stocker.

Le code ada:

```
with Ada.Text_Io;
 with Ada.Numerics.Discrete_Random;

 ..........


   subtype Normal_Character_Type is Character range Character'Val(16#07#) .. Character'Val(16#FF#);

   type Special_Integer_Type is mod 2 ** 22;

    type Random_Value_Of_Character_Array_Type is
    array(Normal_Character_Type'Range) of Special_Integer_Type;

   package Random_Integer_Package is new
   Ada.Numerics.Discrete_Random(Result_Subtype => Special_Integer_Type);

   Random_Value_Of_Character_Array : Random_Value_Of_Character_Array_Type;
```

```
   Random_Integer_Generator        : Random_Integer_Package.Generator;
   --|
   --|       |======|
   --|       | Init |
   --|       |======|
   --|

   --|
   --| create random values associated with every character
   --|
   procedure Init(Seed : Long_Integer := 10009)
      is
      begin
      --|
      --| initialize Randomize_Note_Name with seed
      --|
      Random_Integer_Package.Reset(Gen       => Random_Integer_Generator,
                                   Initiator => Integer(Seed));
      --|
      --| initialize Random_Value_Of_Character_Array() with a random number
      --|
      for K in Normal_Character_Type'Range loop
         Random_Value_Of_Character_Array(K)
         := Random_Integer_Package.Random(Gen => Random_Integer_Generator);
      end loop;

      Number_Of_Collision := 0;
      Number_Of_Hash      := 0;
   end Init;
   --|
   --|       |=========|
   --|       | Hash_Of |
   --|       |=========|
   --|
   function Hash_Of(S               : in     String;
                    Hash_Table_Size : in     Long_Integer := 1021)
      return Long_Integer
      is
      K          : Special_Integer_Type := 0;
      N          : Special_Integer_Type := 0;
      M          : Special_Integer_Type := 1;
      Hash_Value : Long_Integer;
      begin
      --|
      --| an empty string needs not hashing and always returns the same value
      --|
      if S = ""
         then
         raise Hashing_Empty_String;
      end if;
      --|
      --|use the random look up table and add time m
      --|
      for L in S'Range loop
         N := Random_Value_Of_Character_Array(S(L));
         K := K + M * N;
         M := 1 + M;
      end loop;
      --|
      --| apply modulo to keep index in user's range
      --|
      Hash_Value := Long_Integer(K) + Long_Integer(S'Size);
      Hash_Value := Hash_Value mod Hash_Table_Size;
      --|
      --|modulo can return zero which is not a valid index
      --|
      return Hash_Value + 1;
      --|
      --|non text string with strange ascii char  not in Normal_Character_Type'range
      --|
      exception
      when Constraint_Error =>
      raise Hashing_Empty_String;
   end Hash_Of;
```

12.3.2.4 Numérisation par multiplication

Si la clé est déjà numérique, une formule souvent employée est (calculs en réels) :

hash ⟵ ⌊ m (clé A modulo 1) ⌋

Avec ⌊ X ⌋ = partie entière de (x – 0.5)

A : (√5 –1)/2 = 0.6180339887498948502129840487651080138676

Par exemple si clé = 123456

On a pour un tableau de 16381 places (m = 16381 est un nombre premier et c'est IMPORTANT) :

Hashing = ⌊ 16381 (123456 * 0.6180339887498948502129840487651080138676 modulo 1) ⌋

= ⌊ 16381 (76300.004115107018627384150022407993674 28 modulo 1) ⌋

=⌊ 16381 (0.00411510701862738415002240799367428) ⌋
=⌊ 67.40956807823900001630512690553587162867⌋
=66

```
code ada:
    function Hash_By_Multi(Cle        : in     Long_Float;
                           Table_Size : in     Integer)
       return Integer
       is
       A : constant Long_Float := 0.6180339887498948502129840487651080138676;
       M : Long_Float          := Long_Float(Table_Size);
       Z : Long_Float;
       begin
       Z := Cle * A;
       Z := Z - Long_Float(Long_Integer(Z)); -- modulo 1
       Z := M * Z;
       return Integer(Z - 0.5);
    end Hash_By_Multi;
```

12.3.2.5 Numérisation par division

Hashing ⟵ clé modulo m

Extrêmement simple!!! et surtout utilisé pour un re hash après collision.

12.3.2.6 Numérisation quadratique

Surtout utilisée pour les collision
Hash ⟵ (ancienne valeur de hash + $C_1 i + C_2 i^2$) modulo taille du tableau

I : nombre d'essais successifs (après collisions)

12.3.2.7 Numérisation par décalage et manipulation des bits

Cette Méthode donne de très bons résultats:

(H & Z sur 32 bits sans signe)
H ⟵0
Boucle sur tous les caractères de la clé
Décaler H de 4 bits vers la gauche

H ⟵ H + valeur ASCII du caractère courant
Si l'un des 4 bits de poids le plus élevé est égal à 1
Alors
Z ⟵ H
Décaler H de 24 bits vers la droite

H ⟵ H xor Z
Mettre à 0 les 4 bits de poids le plus élevé de H
Fin si
Fin boucle
Renvoi H modulo la taille du tableau

Code ada:

```
    Type index_type is mod .....;-- taille du tableau (nombre premier)


       function H_Of(The_String : in     String)
       return Index_Type
       is
       --|
       --| From Compilers principles, techniques and tools
       --| AHO et all  addison wesley    page 436
       --|
       type H_Type is mod 2 ** 32;
       H : H_Type := H_Type'First;
       Z : H_Type := H_Type'First;
       begin
       for I in The_String'Range loop
          H := H * 2 ** 4;
          H := H + Character'Pos(The_String(I));
```

```
      if (H and 16#F0000000#) /= 0
         then
         Z := H;
         H := H / 2 ** 24;
         H := H xor Z;
         H := H and 16#0FFFFFFF#;
      end if;
   end loop;
   return Index_Type(H mod (1 + H_Type(Index_Type'Last)));
end H_Of;
```

12.3.3 Les collisions et leur résolution

Choisir un nombre premier compris entre 1 et m (environ m/3 par exemple) soit K ce nombre qui est toujours le même!!!

Nouveau hash ⟵ (ancien hash + k) modulo taille du tableau

12.3.4 Table de Hashing: Implémentation avec un tableau simple

Les pseudo code sont écris avec une taille de tableau et un ajout après collision. L'ajout après collision est environ la taille du tableau divisé par 3 (Nombre premiers). Il faut définir la fonction logique =, et les élément **rien** et **déjà utilisé**.

12.3.4.1 Données et Types de données:

Déclaration du générique

```
generic

type Data_Type is private;

Null_Record         : Data_Type;--  rien
Record_Already_Used : Data_Type;--  déjà utilisé

type Index_Type is mod <>;-- necessaire pour les opérations modulo

Increment_After_Collision : Index_Type;-- devra être premier environ la taille du tableau/3

--|
--| for = operator
--|
with function "=" (Left  : in Data_Type;
                   Right : in     Data_Type) return Boolean is <>;
```

Exemple d'instantiation

```
--|
--| index definition for the hash table
--|
type Index_Type is mod 2 ** 18 - 1;-- prime
--|
--|  data as a pointer to a string
--|
type My_Data_Type is access all String;
--|
--| beware : Left may be different from right while Left.all=right.all
--| define = for the hash table algo
--|
function "=" (Left  : in My_Data_Type;
              Right : in     My_Data_Type)
   return Boolean
   is
   begin
   return Left.all = Right.all;
end "=";
--|
--| the algo needs these two to be defined as not in normal use
--|
Null_Record         : constant My_Data_Type := new String'("~");-- rien
Record_Already_Used : constant My_Data_Type := new String'("@");-- déjà utilisé
--|
--| instantiation
--|
package H_Table is new Simple_Hash(Data_Type                => My_Data_Type,
                                   Null_Record              => Null_Record,
                                   Record_Already_Used      => Record_Already_Used,
                                   Increment_After_Collision => 255,-- prime
                                   Index_Type               => Index_Type,
                                   "="                      => "=");
```

Limited private pour **obliger l'utilisateur à utiliser les opérations du package et interdire l'assignation.**

```
package Simple_Hash
  is
  Record_Not_Found_On_Delete            : exception;
  type Hash_Table_Type(Number_Of_Buckets : Index_Type := Index_Type'Last) is limited private;
  type Iterator_Type is limited private;

  ...........

  private
```

La table est déclarée avec un indice et un contenu inconnus à ce stade, ils seront précisés à l'instantiation.

```
   type Hash_Table_Array_Type is array (Index_Type range <>) of Data_Type;
```

La table de hashing est déclarée comme record avec le tableau rempli vide et le nombre d'élément stockés. La variable est déclarée aliased pour permettre de la désigner par un pointeur.

```
   type Hash_Table_Type(Number_Of_Buckets : Index_Type := Index_Type'Last)
      is
      record
      Hash_Table_Array : aliased Hash_Table_Array_Type(0 .. Number_Of_Buckets)
      := (others => Null_Record);
      How_Many_Stored  : Natural   := 0;
   end record;
```

Le type du pointeur sur le tableau

```
   type Hash_Table_Array_Access_Type is access all Hash_Table_Array_Type;
```

Le type de l'itérateur est un record comprenant: le pointeur sur le tableau, le booléen de fin, l'indice courant et le dernier indice du tableau.

```
   type Iterator_Type is
      record
      Done                  : Boolean                     := False;
      Hash_Table_Index      : Index_Type                  := Index_Type'First;
      Number_Of_Buckets     : Index_Type                  := Index_Type'First;
      Hash_Table_Array_Access : Hash_Table_Array_Access_Type := null;
   end record;
   .......
end Simple_Hash;
```

Exemple d'utilisation:

Pour tous les exemples, le calcul de l'indice est intégré à l'appel en utilisant le calcul inclus par défaut h_of(....) dans le programme, mais vous pouvez utiliser le calcul de hashing de votre choix (voir plus haut).

```
   My_Hash_Table           : H_Table.Hash_Table_Type;
   My_Iterator             : H_Table.Iterator_Type;

   ..............
   H_Table.Initialize(Hash_Table => My_Hash_Table);

   H_Table.Add(Data              => new String'("A"),
               To_The_Hash_Table => My_Hash_Table,
               Bucket_Number     => H_Table.H_Of(The_String => "A"));


   H_Table.Seek(Data              => new String'("ABCDE"),
                In_The_Hash_Table => My_Hash_Table,
                Success           => Ok,
                Found_Record      => My_Data,
                Bucket_Number     => H_Table.H_Of(The_String => "ABCDE"));

   H_Table.Delete(Data              => new String'("ABCDE"),
                  In_The_Hash_Table => My_Hash_Table,
                  Bucket_Number     => H_Table.H_Of(The_String => "ABCDE"));


   H_Table.Initialize(Iterator           => My_Iterator,
                      For_The_Hash_Table => My_Hash_Table);
   --|
   --| print all string in hash table
   --|
   while not H_Table.Is_Done(Iterator => My_Iterator) loop
      --|
      --| read and put to screen
      --|
      My_Data := H_Table.Current_Value(Iterator => My_Iterator);
      Ada.Text_Io.Put_Line('>'& My_Data.all & '<');
      --|
      --| to next if any
      --|
      H_Table.To_Next_Value(Iterator => My_Iterator);
   end loop;
```

12.3.4.2 Initialiser

Mettre tout le tableau à rien et initialiser les indices.

12.3.4.3 Ajout

La taille de tableau et l'ajout après collision (environ la taille du tableau divisé par 3) sont tous deux des nombres premiers.

Calcul de la valeur de hash

Hash ← Calculer la valeur de hashing pour la clé
boucle sur la taille du tableau
si Tableau(hash) = rien
ou alors Tableau(hash) = déjà utilisé
tableau(hash)← élément
sortie
fin si
hash ← (ajout après collision +hash) modulo taille de tableau
fin boucle
erreur table pleine

Le code:
Bucket_Number= Calcul de la valeur de hash

```
 procedure Add(Data              : in     Data_Type;
               To_The_Hash_Table : in out Hash_Table_Type;
               Bucket_Number     : in     Index_Type)-- from hash calculation
       is
       Dummy_Bucket_Number : Index_Type := Bucket_Number;
       begin
       for K in To_The_Hash_Table.Hash_Table_Array'Range loop
          if To_The_Hash_Table.Hash_Table_Array(Dummy_Bucket_Number) = Null_Record
             or else To_The_Hash_Table.Hash_Table_Array(Dummy_Bucket_Number) = Record_Already_Used
             then
             To_The_Hash_Table.How_Many_Stored:= To_The_Hash_Table.How_Many_Stored + 1;
             To_The_Hash_Table.Hash_Table_Array(Dummy_Bucket_Number) := Data;
             exit;
          end if;
          Dummy_Bucket_Number       := Dummy_Bucket_Number + Increment_After_Collision;
       end loop;
    end Add;
```

12.3.4.4 Recherche

La recherche se fait sur la chaîne correspondant à l'indice avec deux possibilités: l'élément s'y trouve ou pas!. Si l'élément est trouvé il est retourné. En effet seule la fonction = définie par l'utilisateur sur la partie « clé » des données sert à la recherche. L'élément complet comprenant toutes les données est retourné.
La taille de tableau et l'ajout après collision (environ la taille du tableau divisé par 3) sont tous deux des nombres premiers.

Hash ← Calculer la valeur de hashing pour la clé
boucle sur la taille du tableau
si Tableau(hash) = clé
renvoi vrai
fin si
si Tableau(hash) = rien
renvoi faux
fin si
hash ← (ajout après collision +hash) modulo taille de tableau
fin boucle
renvoi faux

Le code:
Bucket_Number= Calcul de la valeur de hash

```
procedure Seek(Data              : in     Data_Type;
               In_The_Hash_Table : in     Hash_Table_Type;
               Success           :    out Boolean;
               Found_Record      : in out Data_Type;
               Bucket_Number     : in     Index_Type)
       is
       Dummy_Bucket_Number : Index_Type := Bucket_Number;
       begin
       for K in In_The_Hash_Table.Hash_Table_Array'Range loop
          if In_The_Hash_Table.Hash_Table_Array(Dummy_Bucket_Number) = Data
```

```
            then
            Found_Record := In_The_Hash_Table.Hash_Table_Array(Dummy_Bucket_Number);
            Success      := True;
            return;
            elsif In_The_Hash_Table.Hash_Table_Array(Dummy_Bucket_Number) = Null_Record
            then
            Success := False;
            return;
         end if;
         Dummy_Bucket_Number        := Dummy_Bucket_Number + Increment_After_Collision;
      end loop;
      Success := False;
   end Seek;
```

12.3.4.5 Suppression

La suppression est basée sur le fait que l'élément à supprimer EST PRESENT. Si ce n'est pas le cas l'exception Record_Not_Found_On_Delete est utilisée.
La taille de tableau et l'ajout après collision (environ la taille du tableau divisé par 3) sont tous deux des nombres premiers.

Hash ← Calculer la valeur de hashing pour la clé
boucle sur la taille du tableau
si Tableau(hash) contient la clé
tableau(hash)← déjà utilisé
sortie
fin si
hash ← (ajout après collision +hash) modulo taille de tableau
fin boucle

Le code:
Bucket_Number= Calcul de la valeur de hash

```
 procedure Delete(Data             : in     Data_Type;
                  In_The_Hash_Table : in out Hash_Table_Type;
                  Bucket_Number     : in     Index_Type)
      is
      Dummy_Index : Index_Type := Bucket_Number;
      begin
      for K in In_The_Hash_Table.Hash_Table_Array'Range loop
         if In_The_Hash_Table.Hash_Table_Array(Dummy_Index) = Data
            then
            In_The_Hash_Table.Hash_Table_Array(Dummy_Index) := Record_Already_Used;
            In_The_Hash_Table.How_Many_Stored               := In_The_Hash_Table.How_Many_Stored - 1;
            return;
         end if;
         Dummy_Index := Dummy_Index + Increment_After_Collision;
      end loop;
      raise Record_Not_Found_On_Delete;
   end Delete;
```

12.3.4.6 itérateur

Pour utiliser l'itérateur, le tableau sera parcouru jusqu'à trouver un élément valable (ni rien, ni déjà utilisé). L'index correspondant est conservé. Un booléen sert a définir la fin. Il est initialisé à faux. Un pointeur sert a conserver l'accès au tableau.

12.3.4.6.1 Initialiser

```
 procedure Initialize(Iterator           :    out Iterator_Type;
                      For_The_Hash_Table : in out Hash_Table_Type)
      is
      begin
```

Initialisation des variables

```
      Iterator.Done                  := False;
      Iterator.Hash_Table_Index      := 0;
      Iterator.Number_Of_Buckets     := Index_Type'Last;
      Iterator.Hash_Table_Array_Access
      := new Hash_Table_Array_Type'(For_The_Hash_Table.Hash_Table_Array);
```

Rechercher le premier enregistrement

```
      for I in Index_Type'Range loop
         if Iterator.Hash_Table_Array_Access.all (I)     /= Null_Record
```

```
            and Iterator.Hash_Table_Array_Access.all (I) /= Record_Already_Used
            then
            Iterator.Hash_Table_Index := I;
            return;
         end if;
      end loop;
```

Aucun enregistrement trouvé, la structure est vide: mettre le drapeau à vrai

```
      Iterator.Done := True;
end Initialize;
```

12.3.4.6.2 Suivant

```
   procedure To_Next_Value(Iterator : in out Iterator_Type)
      is
```

Passer d'office à l'enregistrement suivant

```
      I : Index_Type := 1 + Iterator.Hash_Table_Index;
      begin
```

Chercher l'enregistrement suivant si il existe

```
      while I in Iterator.Hash_Table_Index .. Index_Type'Last loop
         if Iterator.Hash_Table_Array_Access.all (I) /= Null_Record
            and Iterator.Hash_Table_Array_Access.all (I) /= Record_Already_Used
            then
            Iterator.Hash_Table_Index := I;
            return;
         end if;
         I := 1 + I;
      end loop;
```

Aucun enregistrement trouvé : tous les éléments ont été visités mettre le drapeau à vrai

```
      Iterator.Done := True;

   end To_Next_Value;
```

12.3.4.6.3 Valeur

```
   function Current_Value(Iterator : in     Iterator_Type)
      return Data_Type
      is
      begin
      return Iterator.Hash_Table_Array_Access.all (Iterator.Hash_Table_Index);
   end Current_Value;
```

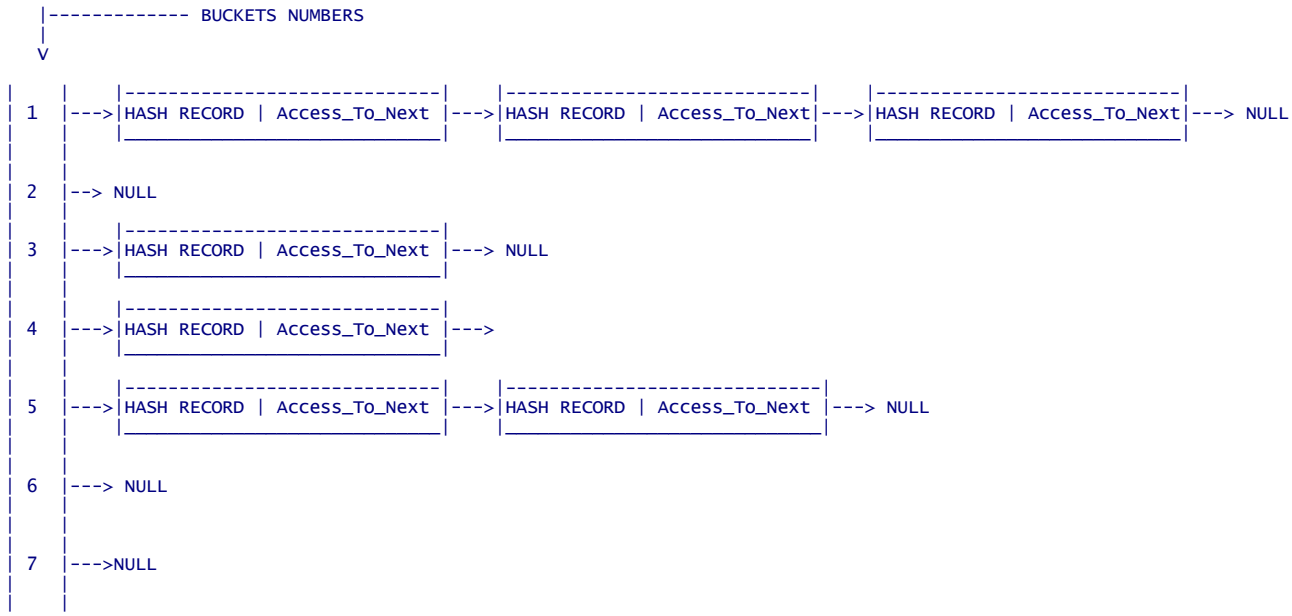
12.3.4.6.4 Fini

```
function Is_Done(Iterator : in     Iterator_Type)
      return Boolean
      is
      begin
      return Iterator.Done;
   end Is_Done;
```

12.3.5 Table de Hashing: Implémentation avec un tableau Chaîné et des pointeurs

Ici un tableau de pointeurs est utilisé. Chaque pointeur peut être vide (null) ou pointer vers un élément composite, formé d'une partie donnée « hash record » et d'une partie pointeur « Access_To_Next ». Chaque position du tableau (ici indice est appelé « bucket number ». Lors de l'ajout, l'indice est calculé par hashing et le nouvel élément introduit avant le premier élément déjà en place. Le concept de collision ne s'applique plus!!. **L'ajout est en temps constant** (du moins en théorie!). La recherche parcours la « chaîne » déjà attachée au contenu du tableau correspondant à la valeur de hashing calculé. La recherche parcours donc en moyenne la moité de la chaîne. **La recherche est en temps proportionnel au rapport du nombre d'éléments à la taille du tableau** (du moins en théorie!).

Voici un exemple de remplissage d'un tel tableau (7 cases pour faciliter le dessin):



12.3.5.1 Données et Types de données:

Déclaration du générique

```
with Ada.Unchecked_Deallocation;
--|
--|
--|  needed for hash_of availaility
--|
generic

type Data_Type is private;
--|
--| for = operator
--|
with function "=" (Left  : in Data_Type;
                   Right : in     Data_Type)
  return Boolean
is <>;

package Linked_Hash
  is
  Record_Not_Found_On_Delete                    : exception;
  type Linked_Hash_Table_Type(Number_Of_Buckets : Positive) is limited private;
  type Iterator_Type is limited private;
```

Exemple d'instantiation:

```
 ........

  procedure Test_Linked_Hash
    is
    --|
    --| index definition for the hash table
    --|
    type Index_Type is mod 2 ** 18 - 1;
    --|
    --|  data as a pointer to a string
    --|
    type My_Data_Type is access all String;
    --|
    --|beware :Left may be different from right while Left.all=right.all
    --| define = for the hash table algo
    --|
    function "=" (Left  : in My_Data_Type;
                  Right : in     My_Data_Type)
       return Boolean
       is
       begin
       return Left.all = Right.all;
    end "=";
    --|
    --| instantiation
    --|
    package H_Table is new Linked_Hash(Data_Type => My_Data_Type,
```

```
                                    "="       => "=");
    --|
    --| general data
    --|
    My_Hash_Table           : H_Table.Linked_Hash_Table_Type(Number_Of_Buckets => 2 ** 10 - 1);
    My_Iterator             : H_Table.Iterator_Type;

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

Limited private: pour **obliger l'utilisateur à utiliser les opérations du package et interdire l'assignation.**

```
    private

    type Node_Type;

    type Node_Access_Type is access all Node_Type;

    type Node_Type is
       record
       Data           : Data_Type;
       Access_To_Next : Node_Access_Type := null;
    end record;
    --|
    --|  used in delete
    --|
    procedure Free_Node is new Ada.Unchecked_Deallocation(Object => Node_Type,
                                                          Name   => Node_Access_Type);
    --|
    --|  hash table
    --|
    type Hash_Table_Array_Type is array (Integer range <>) of Node_Access_Type;
    --|
    --|
    --|
    type Linked_Hash_Table_Type(Number_Of_Buckets : Positive)
       is
       record
       Hash_Table_Array : aliased Hash_Table_Array_Type(0 .. Number_Of_Buckets) := (others => null);
       How_Many_Stored  : Natural                                               := 0;
    end record;
    --|
    --|  needed for the iterator
    --|
    type Hash_Table_Array_Access_Type is access all Hash_Table_Array_Type;
    --|
    --|  iterator
    --|
    type Iterator_Type is
       record
       Current_Node_Access     : Node_Access_Type             := null;
       Hash_Table_Index        : Integer                      := 0;
       Number_Of_Buckets       : Integer                      := 0;
       Hash_Table_Array_Access : Hash_Table_Array_Access_Type := null;
    end record;


  end Linked_Hash;
```

Exemple d'utilisation:

Pour tous les exemples, le calcul de l'indice est intégré à l'appel en utilisant le calcul inclus par défaut h_of(....) dans le programme, mais vous pouvez utiliser le calcul de hashing de votre choix (voir plus haut).

```
    My_Hash_Table           : H_Table.Hash_Table_Type;
    My_Iterator             : H_Table.Iterator_Type;

    ..............
    H_Table.Initialize(Hash_Table => My_Hash_Table);

    H_Table.Add(Data              => new String'("A"),
                To_The_Hash_Table => My_Hash_Table,
                Bucket_Number     => H_Table.H_Of(The_String => "A"));


    H_Table.Seek(Data              => new String'("ABCDE"),
                 In_The_Hash_Table => My_Hash_Table,
                 Success           => Ok,
                 Found_Record      => My_Data,
                 Bucket_Number     => H_Table.H_Of(The_String => "ABCDE"));

    H_Table.Delete(Data              => new String'("ABCDE"),
                   In_The_Hash_Table => My_Hash_Table,
                   Bucket_Number     => H_Table.H_Of(The_String => "ABCDE"));


    H_Table.Initialize(Iterator          => My_Iterator,
                       For_The_Hash_Table => My_Hash_Table);
    --|
    --| print all string in hash table
```

```
   --|
   while not H_Table.Is_Done(Iterator => My_Iterator) loop
      --|
      --| read and put to screen
      --|
      My_Data := H_Table.Current_Value(Iterator => My_Iterator);
      Ada.Text_Io.Put_Line('>'& My_Data.all & '<');
      --|
      --| to next if any
      --|
      H_Table.To_Next_Value(Iterator => My_Iterator);
   end loop;
```

12.3.5.2 Initialiser

Ici, la situation est plus compliquée. Une boucle sur tous les indices du tableau sert a visiter tous les départs possible de chaînes d'éléments. Chaque élément de chaque chaîne est « détruit » l'un après l'autre pour libérer la mémoire correspondante (free_node(...) jusqu'à ce que la chaîne soit vide ci qui s'écrit: null /= Hash_Table.Hash_Table_Array(Local_Bucket_Number). Le nombre d'éléments est mis à zéro.

Pseudo-code:

pointeur ← null
index ← 0
tant que index < longueur du tableau boucle
    tant que table(index) ≠ null boucle
        Faire pointer le pointeur au même endroit que table (index)
        pointeur ← table(index)
        Faire pointer table (index) sur l'élément suivant ou sur vide
        table(index)← pointeur.access_to_next
        Libérer l'espace mémoire
        effacer (pointeur)
        index ←1+ index
    fin boucle
fin boucle
nombre d'élément ← 0

Code ada:

```
   procedure Initialize(Hash_Table : in out Linked_Hash_Table_Type)
      is
      Dummy_Node_Access : Node_Access_Type;
      begin

      for Local_Bucket_Number in Hash_Table.Hash_Table_Array'Range loop
         while null /= Hash_Table.Hash_Table_Array(Local_Bucket_Number) loop

            Dummy_Node_Access := Hash_Table.Hash_Table_Array(Local_Bucket_Number);

            Hash_Table.Hash_Table_Array(Local_Bucket_Number) := Dummy_Node_Access.Access_To_Next;

            Free_Node(Dummy_Node_Access);
         end loop;
      end loop;
      Hash_Table.How_Many_Stored := 0;

   end Initialize;
```

12.3.5.3 Ajout

Il n'existe pas de collision!! chaque élément est mis en tête de chaîne (cela va plus vite). Un nouvel élément est créé puis inséré au début de la chaîne en utilisant une seule instruction. Bucket_Number= Calcul de la valeur de hash.

Pseudo-code:

Calcul index
index ← hash(clé)
Créer dynamiquement un élément contenant data et retourner un pointeur vers cet élément.
pointeur ← créer_élément(data)
Faire pointer le pointeur de l'élément créé ci-dessus au même endroit que table (index)
pointeur.access_to_next ←table(index)
Faire pointer table (index) vers le nouvel élément

table(index) ← pointeur
Libérer l'espace mémoire
effacer (pointeur)
nombre d'éléments ← 1 + nombre d'éléments

Code ada:

```
   procedure Add(Data              : in     Data_Type;
                 To_The_Hash_Table : in out Linked_Hash_Table_Type;
                 Bucket_Number     : in     Natural)
      is
      Local_Bucket_Number : Natural := Bucket_Number mod (1 + To_The_Hash_Table.Number_Of_Buckets);
      begin
      Statistics.Add_Calls := 1 + Statistics.Add_Calls;
      --|
      --| insert
      --|
      To_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number)
      := new Node_Type'(Data           => Data,
                        Access_To_Next => To_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number));

      To_The_Hash_Table.How_Many_Stored := 1 + To_The_Hash_Table.How_Many_Stored;
   end Add;
```

12.3.5.4 Recherche

La recherche se fait sur la chaîne correspondant à l'indice avec deux possibilités: l'élément s'y trouve ou pas!. Si l'élément est trouvé il est retourné. En effet seule la fonction = définie par l'utilisateur sur la partie « clé » des données sert à la recherche. L'élément complet comprenant toutes les données est retourné. Bucket_Number= Calcul de la valeur de hash.

Pseudo-code:

Calcul index
index ← hash(clé)
Faire pointer le pointeur au même endroit que table (index)
pointeur ← table(index)
Boucle pour chercher l'élément si il existe
    tant que pointeur ≠ null boucle
L'opérateur = est spécialement défini pour comparer la clé avec celle contenue dans data
    Si pointeur . Data = clé
        alors
        trouvé ← vrai
        élément ← data
        sortie
    Fin si
Fin boucle
Libérer l'espace mémoire
effacer (pointeur)

Code ada:

```
   procedure Seek(Data              : in     Data_Type;
                  In_The_Hash_Table : in     Linked_Hash_Table_Type;
                  Success           :    out Boolean;
                  Found_Record      : in out Data_Type;
                  Bucket_Number     : in     Natural)
      is
  Local_Bucket_Number : Natural:= Bucket_Number mod (1 + In_The_Hash_Table.Number_Of_Buckets);
  Node_Access : Node_Access_Type := In_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number);
      begin

      Success := False;
      while null /= Node_Access loop
         if Data = Node_Access.all.Data
            then
            Success      := True;
            Found_Record := Node_Access.all.Data;
            Return;
         end if;
         Node_Access := Node_Access.Access_To_Next;
      end loop;
   end Seek;
```

12.3.5.5 Suppression

La suppression est basée sur le fait que l'élément à supprimer EST PRESENT. Si ce n'est pas le cas, l'exception « Record_Not_Found_On_Delete » est utilisée. Bucket_Number= Calcul de la valeur de hash. C'est la première fois qu'il faut **distinguer**

la suppression du **premier élément** de celle du **nième élément**. De plus, il faut deux pointeurs pour éliminer un élément au milieu d'un liste chaînée.

Pseudo-code:

Calcul index
index ← hash(clé)
Faire pointer les pointeurs au même endroit que table (index)
pointeur_précédent ← table(index)
pointeur ← table(index)
Boucle pour chercher l'élément si il existe
tant que pointeur ≠ null boucle
L'opérateur = est spécialement défini pour comparer la clé avec celle contenue dans data
Si pointeur . Data = clé
alors
L'élément à supprimer est trouvé
sortie boucle
Fin si
L'élément à trouver n'est pas celui là
pointeur_précédent ←pointeur
pointeur ← pointeur.suivant
Fin boucle
si pointeur = null
alors
L'élément à trouver n'existe pas! Sortir du programme avec une erreur
sortie erreur
fin si
Surpression du premier élément de la chaîne ??
si pointeur = pointeur.suivant
alors
Modification directe du pointeur du tableau
table(index) ← table(index) .suivant
sinon
pointeur_précédent.suivant ← pointeur.suivant
fin si
Libérer l'espace mémoire
effacer (pointeur)

Code ada:

```
procedure Delete(Data             : in     Data_Type;
                 In_The_Hash_Table : in out Linked_Hash_Table_Type;
                 Bucket_Number     : in     Natural)
    is
    Local_Bucket_Number  : Natural  := Bucket_Number mod (1 + In_The_Hash_Table.Number_Of_Buckets);
    Current_Node_Access:Node_Access_Type := In_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number);
   Previous_Node_Access:Node_Access_Type := In_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number);

    begin

    --| find the item to remove
    while null /= Current_Node_Access loop
       exit when Data = Current_Node_Access.all.Data;
       Previous_Node_Access := Current_Node_Access;
       Current_Node_Access  := Current_Node_Access.Access_To_Next;
    end loop;

    if null = Current_Node_Access
       then
       raise Record_Not_Found_On_Delete;
    end if;
    --| first one on linked list
    if Current_Node_Access = Previous_Node_Access
       then
       In_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number)
       := In_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number).Access_To_Next;
       else
       Previous_Node_Access.Access_To_Next := Current_Node_Access.Access_To_Next;
    end if;
    Free_Node(Current_Node_Access);
    In_The_Hash_Table.How_Many_Stored := In_The_Hash_Table.How_Many_Stored - 1;
 end Delete;
```

12.3.5.6 itérateur

12.3.5.6.1 Initialiser

```
procedure Initialize(Hash_Table : in out Linked_Hash_Table_Type)
   is
   Dummy_Node_Access : Node_Access_Type;
   begin

   for Local_Bucket_Number in Hash_Table.Hash_Table_Array'Range loop
      while null /= Hash_Table.Hash_Table_Array(Local_Bucket_Number) loop

         Dummy_Node_Access := Hash_Table.Hash_Table_Array(Local_Bucket_Number);

         Hash_Table.Hash_Table_Array(Local_Bucket_Number) := Dummy_Node_Access.Access_To_Next;

         Free_Node(Dummy_Node_Access);
      end loop;
   end loop;
   Hash_Table.How_Many_Stored := 0;

end Initialize;
```

12.3.5.6.2 Suivant

```
procedure To_Next_Value(Iterator : in out Iterator_Type)
   is
   procedure Get_Next_Valid_Record(Iterator : in out Iterator_Type)
   is
   begin

   loop
      exit when Iterator.Current_Node_Access /= null;
      Iterator.Hash_Table_Index := 1 + Iterator.Hash_Table_Index;
      exit when Iterator.Hash_Table_Index > Iterator.Number_Of_Buckets;
  Iterator.Current_Node_Access := Iterator.Hash_Table_Array_Access.all (Iterator.Hash_Table_Index);
   end loop;
end Get_Next_Valid_Record;

   begin
   -- go  one step further
   Iterator.Current_Node_Access := Iterator.Current_Node_Access.Access_To_Next;
   -- get next valid record (if any)
   Get_Next_Valid_Record(Iterator => Iterator);
end To_Next_Value;
```

12.3.5.6.3 Fini

```
function Is_Done(Iterator : in     Iterator_Type)
   return Boolean
   is
   begin
   return Iterator.Hash_Table_Index > Iterator.Number_Of_Buckets;
end Is_Done;
```

12.3.5.6.4 Valeur

```
function Current_Value(Iterator : in     Iterator_Type)
       return Data_Type
       is
       begin
       return Iterator.Current_Node_Access.Data;
    end Current_Value;
```

12.4 LE STACK

C'est une structure de type LIFO (last in first out). Son implémentation dans un tableau est simple: un tableau et un indice pointant sur la valeur la plus récente.
Il y a deux choix possibles:

- Incrémenter l'indice avant l'assignation.
- **Incrémenter l'indice après l'assignation (c'est l'option choisie ici).**

12.4.0.1 Données et Types de données:

La partie générique

```
generic

type Data_Type is private;
Stack_Size : Positive;


package Simple_Stack
  is

  Stack_Full  : exception;
  Stack_Empty : exception;
```

Limited private: pour **obliger l'utilisateur à utiliser les opérations du package et interdire l'assignation.**

```
   type Iterator_Type is limited private;
   type Stack_Type is limited private;



   private
   --|
   --|  stack table
   --|
   type Stack_Table_Array_Type is array (Positive range <>) of Data_Type;
   --|
   --|
   --|
   type Stack_Type(Size : Positive := Stack_Size)
      is
      record
      Stack_Table_Array : aliased Stack_Table_Array_Type(1 .. Stack_Size);
      Current_Index     : Positive := 1;
   end record;
   --|
   --|  needed for the iterator
   --|
   type Stack_Table_Array_Access_Type is access all Stack_Table_Array_Type;
   --|
   --|  iterator
   --|
   type Iterator_Type is
      record
      Stack_Table_Index        : Integer                        := 0;
      Stack_Table_Array_Access : Stack_Table_Array_Access_Type := null;
   end record;

   ...........

end Simple_Stack;
```

Exemple d'instantiation:

```
.........
  --|
  --|  data as a pointer to a string
  --|
  type My_Data_Type is access all String;
  --|
  --| size definition for the stack
  --|
  How_Many_Records : constant := 2 ** 19;
  --|
  --| instantiation
  --|
  package My_Stack is new Simple_Stack(Data_Type  => My_Data_Type,
                                       Stack_Size => How_Many_Records);

  --|
  --| general data
  --|
  A_Stack                : My_Stack.Stack_Type;
  My_Iterator            : My_Stack.Iterator_Type;
........;

end Test_Simple_Stack;
```

Exemple d'utilisation:

```
  ............
   My_Stack.Initialize(The_Stack => A_Stack);

   My_Stack.Push(Data         => new String'("A"),
                 In_The_Stack => A_Stack);
```

```
   --|=================
   --|  iterator  test
   --|=================
   --|
   My_Stack.Initialize(Iterator      => My_Iterator,
                       For_The_Stack => A_Stack);


   while not My_Stack.Is_Done(Iterator => My_Iterator) loop
      --|
      --| read and put to screen
      --|
      My_Data := My_Stack.Current_Value(Iterator => My_Iterator);
      Ada.Text_Io.Put_Line('>'& My_Data.all & '<');
      --|
      --| to next if any
      --|
      My_Stack.To_Next_Value(Iterator => My_Iterator);
   end loop;


      My_Stack.Pop(The_Stack => A_Stack);
.........
```

12.4.1 Le stack (pile) : Implémentation avec un tableau

Ici, l'indice est incrémenté ***après*** l'assignation. Le code de l'ajout, de la délétion, de la valeur et de l'initialisation en tiennent compte. Un code tout aussi valable peut être écrit en utilisant l'assignation après l'incrémentation mais est laissé au lecteru comme exercice.

12.4.1.1 Ajout (push)

Grande simplicité du code : incrémentation après l'assignation.

```
   procedure Push(Data         : in     Data_Type;
                  In_The_Stack : in out Stack_Type)
      is
      begin
      In_The_Stack.Stack_Table_Array(In_The_Stack.Current_Index) := Data;
      In_The_Stack.Current_Index := 1 + In_The_Stack.Current_Index;
      exception
      when Constraint_Error => raise Stack_Full;
   end Push;
```

12.4.1.2 Délétion (pop)

Notez la simplicité du code: on “oublie” le dernier élément entré en ajoutant -1 à l'indice.

```
   procedure Pop(The_Stack : in out Stack_Type)
      is
      begin
      The_Stack.Current_Index := The_Stack.Current_Index - 1;

      exception
      when Constraint_Error => raise Stack_Empty;
   end Pop;
```

12.4.1.3 Dernière valeur (top)

Renvoi la dernière valeur entrée.

```
   function Top_Of(The_Stack : in     Stack_Type)
      return Data_Type
      is
      begin
      return The_Stack.Stack_Table_Array(The_Stack.Current_Index - 1);
      exception
      when Constraint_Error => raise Stack_Empty;
   end Top_Of;
```

12.4.1.4 Initialiser

Notez la simplicité du code: on “oublie” tous les éléments entrés en mettant l'indice à 1;

```
   procedure Initialize(The_Stack : in out Stack_Type)
```

```
      is
      begin
      The_Stack.Current_Index := 1;
   end Initialize;
```

12.4.1.5 itérateur

Tous ces codes sont d'une grande simplicité!

Initialiser

```
   procedure Initialize(Iterator      :    out Iterator_Type;
                        For_The_Stack : in     Stack_Type)
      is

      begin
      Iterator := (Stack_Table_Index => For_The_Stack.Current_Index - 1,
            Stack_Table_Array_Access => new Stack_Table_Array_Type'(For_The_Stack.Stack_Table_Array));

   end Initialize;
```

Valeur

```
   function Current_Value(Iterator : in     Iterator_Type)
      return Data_Type
      is
      begin
      return Iterator.Stack_Table_Array_Access.all (Iterator.Stack_Table_Index);
   end Current_Value;
```

Suivant

```
   procedure To_Next_Value(Iterator : in out Iterator_Type)
      is
      begin
      Iterator.Stack_Table_Index := Iterator.Stack_Table_Index - 1;
   end To_Next_Value;
```

Fini

```
   function Is_Done(Iterator : in     Iterator_Type)
      return Boolean
      is
      begin
      return Iterator.Stack_Table_Index = 0;
   end Is_Done;
```

12.4.2 Le stack: Implémentation avec des pointeurs

Une liste chaînée du même type que celle utilisée pour la table de hashing sert pour l'implémentation.

12.4.2.1 Données et Types de données:

La partie générique:

```
generic

  type Data_Type is private;


  package Linked_Stack
    is


    Stack_Empty : exception;

    type Iterator_Type is limited private;
  type Stack_Type is limited private;
```

Limited private: pour **obliger l'utilisateur à utiliser les opérations du package et interdire l'assignation.**

```
...........
  private

    type Node_Type;
    --|
    --|
    --|
    type Node_Access_Type is access all Node_Type;
    --|
    --|
    --|
    type Node_Type is
       record
       Data           : Data_Type;
       Access_To_Next : Node_Access_Type := null;
    end record;
    --|
    --|  used in pop
    --|
```

Cette instantiation, que vous retrouverez dans tous les programmes avec pointeurs sert à récupérer la memoire quant o détruit l'objet pointé et le pointeur.

```
    procedure Free_Node is new Ada.Unchecked_Deallocation(Object => Node_Type,
                                                          Name   => Node_Access_Type);

    --|
    --|the stack itself
    --|
    type Stack_Type
       is
       record
       Node_Access     : Node_Access_Type := null;
       How_Many_Stored : Natural          := 0;
    end record;

    --|
    --|  iterator
    --|
    type Iterator_Type is
       record
       Current_Node_Access : Node_Access_Type := null;
    end record;
.............
```

Exemple d'instantiation:

```
    --|
    --|  data as a pointer to a string
    --|
    type My_Data_Type is access all String;
    --|
    --| instantiation
    --|
    package My_Stack is new Linked_Stack(Data_Type => My_Data_Type);

    --|
    --| general data
    --|
    A_Stack                 : My_Stack.Stack_Type;
    My_Iterator             : My_Stack.Iterator_Type;
```

Exemples d'utilisation:

```
    --|
    --|  init table data structures
    --|
    My_Stack.Initialize(The_Stack => A_Stack);

    My_Stack.Push(Data         => new String'("A"),
                  In_The_Stack => A_Stack);


    My_Stack.Initialize(Iterator      => My_Iterator,
                        For_The_Stack => A_Stack);
    --|
    --| print all string in hash table
    --|
    while not My_Stack.Is_Done(Iterator => My_Iterator) loop
       --|
       --| read and put to screen
       --|
       My_Data := My_Stack.Current_Value(Iterator => My_Iterator);
       Ada.Text_Io.Put_Line('>'& My_Data.all & '<');
       --|
       --| to next if any
       --|
       My_Stack.To_Next_Value(Iterator => My_Iterator);
    end loop;
```

```
My_Stack.Pop(The_Stack => A_Stack);
```

12.4.2.2 Ajout (push)

Pseudo-code:

Créer dynamiquement un élément contenant data et retourner un pointeur vers cet élément.
pointeur ← créer_élément(data)
Faire pointer le pointeur de l'élément créé ci-dessus au même endroit que le stack
pointeur.access_to_next ← stack
Faire pointer le stack vers le nouvel élément
stack ← pointeur
CES TROIS INSTRUCTIONS EN ADA (une seule instruction regroupe toutes ces actions):
In_The_Stack.Node_Access := new Node_Type'(Data => Data,
Access_To_Next => In_The_Stack.Node_Access);
Libérer l'espace mémoire
effacer (pointeur)
nombre d'éléments ← 1 + nombre d'éléments

Code ada:

```
procedure Push(Data         : in     Data_Type;
               In_The_Stack : in out Stack_Type)
   is
   begin
   In_The_Stack.How_Many_Stored := 1 + In_The_Stack.How_Many_Stored;
   In_The_Stack.Node_Access     := new Node_Type'(Data       => Data,
                                                  Access_To_Next => In_The_Stack.Node_Access);
end Push;
```

12.4.2.3 Délétion (pop)

Pseudo-code:

pointeur ← vide
Faire pointer le pointeur au même endroit que le stack
pointeur.access_to_next ← stack
Avancer le stack vers l'élément suivant
stack ← pointeur
Libérer l'espace mémoire
effacer (pointeur)
nombre d'éléments ← 1 + nombre d'éléments

Code ada:

```
procedure Pop(The_Stack : in out Stack_Type)
   is
   Z : Node_Access_Type := null;
   begin
   Z                          := The_Stack.Node_Access;
   The_Stack.Node_Access      := The_Stack.Node_Access.Access_To_Next;
   The_Stack.How_Many_Stored := The_Stack.How_Many_Stored - 1;

   Free_Node(X => Z);

end Pop;
```

12.4.2.4 Dernière valeur (top)

```
function Top_Of(The_Stack : in     Stack_Type)
       return Data_Type
       is
       begin
       return The_Stack.Node_Access.Data;
end Top_Of;
```

12.4.2.5 Initialiser

Remarquer l'ordre des instructions: **Il ne faut pas perdre le pointeur «z»** !

Pseudo-code:

pointeur ← stack
Boucle tant que tous les éléments ne sont pas supprimés
tant que pointeur ≠ null boucle
effacer (pointeur)
stack ← stack.suivant
fin boucle
nombre d'éléments ← 1 + nombre d'éléments

Code ada:

```
procedure Initialize(The_Stack : in out Stack_Type)
   is
   Z : Node_Access_Type := The_Stack.Node_Access;
   begin

   while null /= Z loop

      Free_Node(X => Z);
      The_Stack.Node_Access := The_Stack.Node_Access.Access_To_Next;
      Z                     := The_Stack.Node_Access;

   end loop;

   The_Stack.How_Many_Stored := 0;

end Initialize;
```

12.4.2.6 Itérateur

12.4.2.6.1 Initialiser

Le pointeur visitera les éléments dans l'ordre d'un stack: le plus récent en premier.

```
procedure Initialize(Iterator      :    out Iterator_Type;
                     For_The_Stack : in     Stack_Type)
   is
   begin
   Iterator.Current_Node_Access := For_The_Stack.Node_Access;
end Initialize;
```

12.4.2.6.2 Valeur

Renvoi simplement les données courantes

```
function Current_Value(Iterator : in     Iterator_Type)
   return Data_Type
   is
   begin
   return Iterator.Current_Node_Access.Data;
end Current_Value;
```

12.4.2.6.3 Suivant

Le pointeur avance d'un cran.

```
procedure To_Next_Value(Iterator : in out Iterator_Type)
   is
   begin
   Iterator.Current_Node_Access := Iterator.Current_Node_Access.Access_To_Next;
end To_Next_Value;
```

12.4.2.6.4 Fini

Quant le pointeur est vide: le parcours est terminé.

```
    function Is_Done(Iterator : in     Iterator_Type)
       return Boolean
       is
       begin
       return Iterator.Current_Node_Access = null;
    end Is_Done;
```

12.5 LA QUEUE

C'est une structure de type FIFO (first in first out).

12.5.1 La queue: Implémentation avec un tableau

La queue est la première structure dont la complexité de l'implémentation impose une étude approfondie du fonctionnement. L'utilisation d'un tableau circulaire se compose d'un tableau ordinaire dont l'indice est d'un type modulo (comme pour la table de hashing). La complexité est introduite par l'utilisation de 2 indices: un pour le début de la queue et un pour la fin de la queue. Par définition, les nouveaux éléments sont ajoutés à l fin de la queue et les éléments sont lus ou supprimés au début de la queue. La gestion d'erreurs se fait par des exceptions. Le calcul du nombre d'éléments stockés dans la structure est plus compliqué. Il était simpliste jusqu'à présent. Ici, l'indice est incrémenté ***après*** l'assignation. Le code de l'ajout, de la délétion, de la valeur et de l'initialisation en tiennent compte.

12.5.1.1 Données et Types de données:

La partie générique:

```
  generic

  type Data_Type is private;

  type Index_Type is mod <>;

  package Simple_Queue
    is

    Queue_Full  : exception;
    Queue_Empty : exception;

    type Iterator_Type is limited private;
    type Queue_Type is limited private;
```

Limited private: pour **obliger l'utilisateur à utiliser les opérations du package et interdire l'assignation.**

```
...........
    private
    --|
    --|  queue table
    --|
    type Queue_Table_Array_Type is array (Index_Type range <>) of Data_Type;
    --|
    --| declare queue
    --|
    type Queue_Type
       is
       record
       Queue_Table_Array : aliased Queue_Table_Array_Type(Index_Type'Range);
       Head_Index        : Index_Type := 0;
       Tail_Index        : Index_Type := 0;
    end record;
    --|
    --|  needed for the iterator
    --|
    type Queue_Table_Array_Access_Type is access all Queue_Table_Array_Type;
    --|
    --|  iterator
    --|
    type Iterator_Type is
       record
```

```
      Current_Iterator_Index   : Index_Type                   := 0;
      Queue_Iterator_Index     : Index_Type                   := 0;
      Queue_Table_Array_Access : Queue_Table_Array_Access_Type := null;
   end record;
.........
```

Example d'instantiation:

```
   --|
   --|  data as a pointer to a string
   --|
   type My_Data_Type is access all String;
   --|
   --| size definition for the Queue
   --|
   How_Many_Records : constant := 2 ** 19;
   type Index_Type is mod How_Many_Records;
   --|
   --| instantiation
   --|
   package My_Queue is new Simple_Queue(Data_Type  => My_Data_Type,
                                        Index_Type => Index_Type);

   --|
   --| general data
   --|
   A_Queue                 : My_Queue.Queue_Type;
   My_Iterator             : My_Queue.Iterator_Type;
```

Example d'utilisation:

```
   My_Queue.Initialize(The_Queue => A_Queue);


   My_Queue.Add_To_Tail(Data         => new String'("ABCDE"),
                        To_The_Queue => A_Queue);

   My_Queue.Initialize(Iterator      => My_Iterator,
                       For_The_Queue => A_Queue);
   --|
   --| print all strings in Queue
   --|
   while not My_Queue.Is_Done(Iterator => My_Iterator) loop
      --|
      --| read and put to screen
      --|
      My_Data := My_Queue.Current_Value(Iterator => My_Iterator);
      Ada.Text_Io.Put_Line('>'& My_Data.all & '<');
      --|
      --| to next if any
      --|
      My_Queue.To_Next_Value(Iterator => My_Iterator);
   end loop;

      My_Queue.Remove_From_Head(The_Queue => A_Queue);
.........
```

12.5.1.2 Ajout

Pseudo-code:

Test tableau plein
Si Tail_Index + 1 = Head_Index
alors
sortie erreur tableau plein
fin si
Assignation
tableau(Tail_Index) ← data
incrément de l'indice de queue
Tail_Index ← 1+ Tail_Index

Code ada:

```
   procedure Add_To_Tail(Data         : in     Data_Type;
                         To_The_Queue : in out Queue_Type)
      is
      begin

      if To_The_Queue.Tail_Index + 1 = To_The_Queue.Head_Index
         then
         raise Queue_Full;
```

```
      end if;

      To_The_Queue.Queue_Table_Array(To_The_Queue.Tail_Index) := Data;
      To_The_Queue.Tail_Index                                 := 1 + To_The_Queue.Tail_Index;
   end Add_To_Tail;
```

12.5.1.3 Délétion

Pseudo-code:

Test tableau vide
Si Tail_Index = Head_Index
alors
sortie erreur tableau vide
fin si
incrément de l'indice pour « oublier » l'élément
Head_Index ← 1+ Head_Index

Code ada:

```
   procedure Remove_From_Head(The_Queue : in out Queue_Type)
      is
      begin

      if The_Queue.Tail_Index = The_Queue.Head_Index
         then
         raise Queue_Empty;
      end if;

      The_Queue.Head_Index := 1 + The_Queue.Head_Index;

   end Remove_From_Head;
```

12.5.1.4 Dernière valeur

```
   function Head_Of(The_Queue : in     Queue_Type)
      return Data_Type
      is
      begin
      return The_Queue.Queue_Table_Array(The_Queue.Head_Index);
   end Head_Of;
```

12.5.1.5 Initialiser

Attention, head=tail veut dire tableau vide et Tail + 1 = Head_ tableau plein. Il y a donc une case de perdue!!!.

```
   procedure Initialize(The_Queue : in out Queue_Type)
      is
      begin
      The_Queue.Tail_Index := The_Queue.Head_Index;
   end Initialize;
```

12.5.1.6 Itérateur

12.5.1.6.1 Initialiser

Création des variables utilisées par l'itérateur et initialisation de l'index de début, de fin et du pointeur vers la table qui contient les données.

```
   procedure Initialize(Iterator      :    out Iterator_Type;
                        For_The_Queue : in     Queue_Type)
      is

      begin

      Iterator := (Current_Iterator_Index   => For_The_Queue.Head_Index,
                   Queue_Iterator_Index     => For_The_Queue.Tail_Index,
   Queue_Table_Array_Access => new Queue_Table_Array_Type'(For_The_Queue.Queue_Table_Array));

   end Initialize;
```

12.5.1.6.2 Valeur

Renvoi de la valeur courante.

```
   function Current_Value(Iterator : in     Iterator_Type)
      return Data_Type
      is
      begin
      return Iterator.Queue_Table_Array_Access.all (Iterator.Current_Iterator_Index);
   end Current_Value;
```

12.5.1.6.3 Suivant

Incrémentation de l'index.

```
  procedure To_Next_Value(Iterator : in out Iterator_Type)
       is
       begin
       Iterator.Current_Iterator_Index := 1  Iterator.Current_Iterator_Index;
    end To_Next_Value;
```

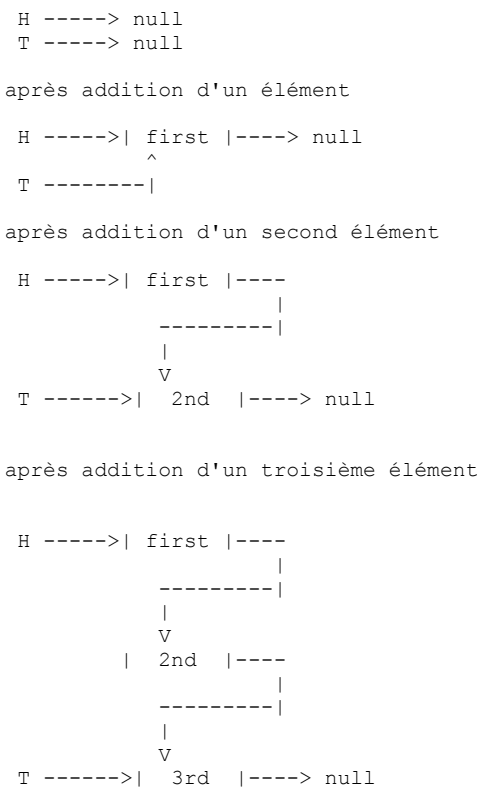
12.5.1.6.4 Fini

Test si courant = fin.

```
 function Is_Done(Iterator : in     Iterator_Type)
       return Boolean
       is
       begin
       return Iterator.Current_Iterator_Index = Iterator.Queue_Iterator_Index;
    end Is_Done;
```

12.5.2 La queue: Implémentation avec des pointeurs

La queue est implémentée avec deux pointeurs H et T, La queue vide, la queue avec un seul élément et la queue avec n éléments est illustrée ci-dessous. C'est une liste chaînée attachée au pointeur H avec comme particularité que l'élément le plus récent est ajouté en bout de chaîne et est pointé par le pointeur T pour pouvoir y accéder rapidement dans le cas d'un ajout. Le pointeur H sert à la lecture de l'élément le plus ancien et à sa suppression. Comme dans toutes les structures avec pointeurs, le nombre d'objets est conservé séparément.



12.5.2.1 Données et Types de données:

La partie générique:

```
with Ada.Unchecked_Deallocation;

generic

type Data_Type is private;


package Linked_Queue
  is


  Queue_Empty : exception;

  type Iterator_Type is limited private;
type Queue_Type is limited private;
```

Limited private: pour **obliger l'utilisateur à utiliser les opérations du package et interdire l'assignation.**

```
  private

  type Node_Type;
  --|
  --|
  --|
  type Node_Access_Type is access all Node_Type;
  --|
  --|
  --|
  type Node_Type is
     record
     Data           : Data_Type;
     Access_To_Next : Node_Access_Type := null;
  end record;
  --|
  --|  used in deletions
  --|
  procedure Free_Node is new Ada.Unchecked_Deallocation(Object => Node_Type,
                                                        Name   => Node_Access_Type);
```

```
   --|
   --|the queue itself
   --|
   type Queue_Type
      is
      record
      Head_Node_Access : Node_Access_Type := null;
      Tail_Node_Access : Node_Access_Type := null;
      How_Many_Stored  : Natural          := 0;
   end record;

   --|
   --|  iterator
   --|
   type Iterator_Type is
      record
      Current_Node_Access : Node_Access_Type := null;
   end record;
```

Exemple d'instantiation:

```
   --|
   --|  data as a pointer to a string
   --|
   type My_Data_Type is access all String;
   --|
   --| size definition for the Queue
   --|
   How_Many_Records : constant := 2 ** 20;
   --|
   --| instantiation
   --|
   package My_Queue is new Linked_Queue(Data_Type => My_Data_Type);
   --|
   --| general data
   --|
   A_Queue                 : My_Queue.Queue_Type;
   My_Iterator             : My_Queue.Iterator_Type;
```

Exemples d'utilisation:

```
My_Queue.Add_To_Tail(Data         => new String'("ABCDE"),
                     To_The_Queue => A_Queue);

   --|
   --|=================
   --|  interator  test
   --|=================
   --|
   Ada.Text_Io.Put_Line("iterator");

   My_Queue.Initialize(Iterator      => My_Iterator,
                       For_The_Queue => A_Queue);
   --|
   --| print all strings in Queue
   --|
   while not My_Queue.Is_Done(Iterator => My_Iterator) loop
      --|
      --| read and put to screen
      --|
      My_Data := My_Queue.Current_Value(Iterator => My_Iterator);
      Ada.Text_Io.Put_Line('>'& My_Data.all & '<');
      --|
      --| to next if any
      --|
      My_Queue.To_Next_Value(Iterator => My_Iterator);
   end loop;

  My_Queue.Remove_From_Head(The_Queue => A_Queue);
```

12.5.2.2 Ajout

Ici, le mécanisme d'addition est différent pour le premier élément de celui pour l'élément numéro n:

```
Départ vide

  H -----> null
  T -----> null

ajout du premier élément

  H ----->| first |----> null
          ^
  T --------|

ajout du second élément

 H ----->| first |----
                    |
           ---------|
           |
           V
 T ------>|  2nd  |----> null


ajout du troisième élément


 H ----->| first |----
                    |
           ---------|
           |
           V
         |  2nd  |----
                    |
           ---------|
           |
           V
 T ------>|  3rd  |----> null

et ainsi de suite
```

```
procedure Add_To_Tail(Data         : in     Data_Type;
                      To_The_Queue : in out Queue_Type)
   is
   begin


   if 0 = To_The_Queue.How_Many_Stored
      then
      --
      -- ======= from
      --
      -- H -----> null
      -- T -----> null
      --
      -- ======= to
      --
      -- H -----> first |----> null
      --          ^
      -- T --------|
      --
      To_The_Queue.Head_Node_Access := new Node_Type'(Data           => Data,
                                                      Access_To_Next => null);
      To_The_Queue.Tail_Node_Access := To_The_Queue.Head_Node_Access;

      else
      --
      --========== starting at  ==========
      --
      -- H -----> first |----
      --                  |
      --           ---------|
      --           |
      --           V
      -- T -----> 2nd  |----> null
      --
      --========== after 1 st step  ==========
      --
      --
      -- H -----> first |----
      --                  |
      --           ---------|
      --           |
      --           V
      -- T -----> 2nd  |---
      --                  |
      --           ---------|
      --           |
```

```
      --            V
      --          |  3rd  |----> null
      --
      To_The_Queue.Tail_Node_Access.Access_To_Next := new Node_Type'(Data           => Data,
                                                                     Access_To_Next => null);

      --
      --========== after 2 nd step  ==========
      --
      -- H ----->| first |----
      --                   |
      --            ---------|
      --            |
      --            V
      --          |  2nd  |----
      --                   |
      --            ---------|
      --            |
      --            V
      -- T ----->|  3rd  |----> null
      --
      --
      To_The_Queue.Tail_Node_Access := To_The_Queue.Tail_Node_Access.Access_To_Next;
   end if;

   To_The_Queue.How_Many_Stored := 1  To_The_Queue.How_Many_Stored;

end Add_To_Tail;
```

12.5.2.3 Délétion

Ici, le mécanisme de suppression est différent pour le dernier élément de celui pour l'élément numéro n:

```
procedure Remove_From_Head(The_Queue : in out Queue_Type)
   is
   Z : Node_Access_Type := null;
   begin

   case The_Queue.How_Many_Stored
      is
      when 0 => raise Queue_Empty;

      when 1 =>
      --
      -- ======= from
      --
      -- H ----->| first |----> null
      --          ^
      -- T --------|
      --
      --
      -- ======= to
      --
      -- H -----> null
      -- T -----> null
      --
      Z                          := The_Queue.Head_Node_Access;
      The_Queue.Head_Node_Access := null;
      The_Queue.Tail_Node_Access := null;

      Free_Node(X => Z);


      when others =>
      --
      --============ from
      -- H ----->| first |----
      --                   |
      --            ---------|
      --            |
      --            V
      --          |  2nd  |----
      --                   |
      --            ---------|
      --            |
      --            V
      -- T ----->|  3rd  |----> null
      --
      --
      --
      --========== to  ==========
      --
      -- H ----->|  2nd  |----
      --                    |
      --            ---------|
      --            |
      --            V
      -- T ----->|  3rd  |----> null
      --
      --
      Z                          := The_Queue.Head_Node_Access;
      The_Queue.Head_Node_Access := The_Queue.Head_Node_Access.Access_To_Next;
      Free_Node(X => Z);
   end case;
```

```
      The_Queue.How_Many_Stored := The_Queue.How_Many_Stored - 1;

   end Remove_From_Head;
```

12.5.2.4 Dernière valeur

Renvoi la valeur courante.

```
   function Head_Of(The_Queue : in     Queue_Type)
      return Data_Type
      is
      begin
      return The_Queue.Head_Node_Access.Data;
   end Head_Of;
```

12.5.2.5 Initialiser

Noter l'utilisation de la procédure de suppression décrite plus haut. Cette méthode est à recommander à chaque fois que son utilisation est possible.

```
   procedure Initialize(The_Queue : in out Queue_Type)
      is
      begin
      while The_Queue.How_Many_Stored /= 0 loop
         Remove_From_Head(The_Queue => The_Queue);
      end loop;
   end Initialize;
```

12.5.2.6 Itérateur

12.5.2.6.1 Initialiser

Fait pointer le pointeur auxiliaire vers le premier élément de la queue (le pus ancien).

```
   procedure Initialize(Iterator      :    out Iterator_Type;
                        For_The_Queue : in     Queue_Type)
      is
      begin
      Iterator.Current_Node_Access := For_The_Queue.Head_Node_Access;
   end Initialize;
```

12.5.2.6.2 Valeur

Renvoi la valeur courante.

```
   function Current_Value(Iterator : in     Iterator_Type)
      return Data_Type
      is
      begin
      return Iterator.Current_Node_Access.Data;
   end Current_Value;
```

12.5.2.6.3 Suivant

Fait avancer d'une position le pointeur auxiliaire.

```
   procedure To_Next_Value(Iterator : in out Iterator_Type)
      is
      begin
      Iterator.Current_Node_Access := Iterator.Current_Node_Access.Access_To_Next;
   end To_Next_Value;
```

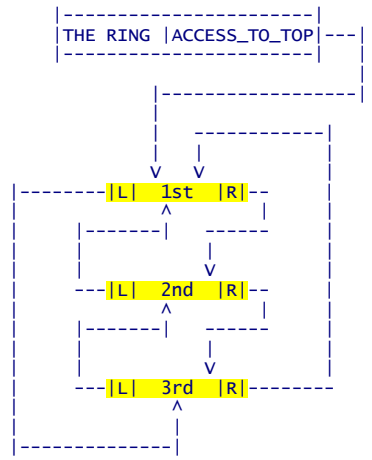
12.5.2.6.4 Fini

Est vrai quant le pointeur auxiliaire est vide.

```
    function Is_Done(Iterator : in     Iterator_Type)
       return Boolean
       is
       begin
       return Iterator.Current_Node_Access = null;
    end Is_Done;
```

12.6 L'ANNEAU

C'est une structure totalement symétrique. Il est possible de se déplacer d'un cran vers la gauche ou la droite de l'élément courant. Cet élément courant est le seul directement accessible. Un nouvel élément peut être inséré/supprimé à gauche ou à droite de l'élément courant. Les codes correspondant sont donc eux aussi symétriques. Chaque élément comprend : un pointeur gauche (L), un élément (1st, 2nd, 3rd, ...) et un pointeur droit (R). Voici une représentation d'un anneau avec 3 éléments. L'anneau lui même est une structure composite comprenant, entre autres, le pointeur sur l'élément courant.



12.6.1 L'anneau: Implémentation avec des pointeurs

La partie générique:

```
.....

  generic

  type Data_Type is private;


  package Linked_Ring
    is

    Ring_Empty     : exception;
    Bookmark_Empty : exception;

    type Iterator_Type is limited private;
    type Ring_Type is limited private;

  type Direction_Type is (Left_Sided, Right_Sided);
```

Limited private: pour **obliger l'utilisateur à utiliser les opérations du package et interdire l'assignation.**

```
private
    --|
    --|
    --|
    type Node_Type;
    --|
    --|
    --|
    type Node_Access_Type is access all Node_Type;
    --|
    --|
    --|
    type Ring_Type is
       record
```

```
   Access_To_Top      : Node_Access_Type := null;
   Access_To_Bookmark : Node_Access_Type := null;
   How_Many_Stored    : Integer          := 0;
end record;
--|
--|
--|
type Node_Type is
   record
   Data  : Data_Type;
   Left  : Node_Access_Type := null;
   Right : Node_Access_Type := null;
end record;

type Iterator_Type is
   record
   Current_Node_Access : Node_Access_Type;
   Count               : Integer        := 0;
   Count_Until         : Integer        := 0;
   Side                : Direction_Type := Direction_Type'First;
end record;
--|
--|  used in deletions
--|
procedure Free_Node is new Ada.Unchecked_Deallocation(Object => Node_Type,
                       Name  => Node_Access_Type);
```

Exemple **d'instantiation:**

```
--|
--|  data as a pointer to a string
--|
type My_Data_Type is access all String;
--|
--| size definition for the RING
--|
How_Many_Records : constant := 2 ** 20;
--|
--| instantiation
--|
package My_Ring is new Linked_Ring(Data_Type => My_Data_Type);
--|
--| general data
--|
A_Ring                  : My_Ring.Ring_Type;
My_Iterator             : My_Ring.Iterator_Type;
```

Exemple d'utilisation:

```
--|
--|  init table data structures
--|
My_Ring.Clear(The_Ring => A_Ring);
--|
--|  add    strings
--|
My_Ring.Insert(The_Item    => new String'("A"),
               In_The_Ring => A_Ring);
```

Noter l'utilisation du paramètre par défaut décrit dans le code de la procédure insert:
Direction : in Direction_Type := Left_Sided
Voir la définition de cet appel dans le code ci-dessous.

```
My_Ring.Insert(The_Item    => new String'("ABCDE"),
               In_The_Ring => A_Ring);

--|
--|=================
--|  interator  test
--|=================
--|


My_Ring.Initialize(Iterator     => My_Iterator,
                   For_The_Ring => A_Ring);
--|
--| print all strings in RING
--|
while not My_Ring.Is_Done(Iterator => My_Iterator) loop
   --|
   --| read and put to screen
   --|
   My_Data := My_Ring.Current_Value(Iterator => My_Iterator);
   Ada.Text_Io.Put_Line('>'& My_Data.all & '<');
   --|
```

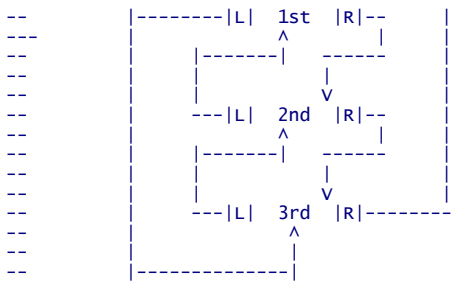
```
      --| to next if any
      --|
      My_Ring.To_Next_Value(Iterator => My_Iterator);
   end loop;


      My_Ring.Delete_Top(In_The_Ring => A_Ring);
```

Même utilisation du paramètre par défaut décrit dans le code de la procédure delete:
Direction : in Direction_Type := Left_Sided
Voir la définition de cet appel dans le code ci-dessous.

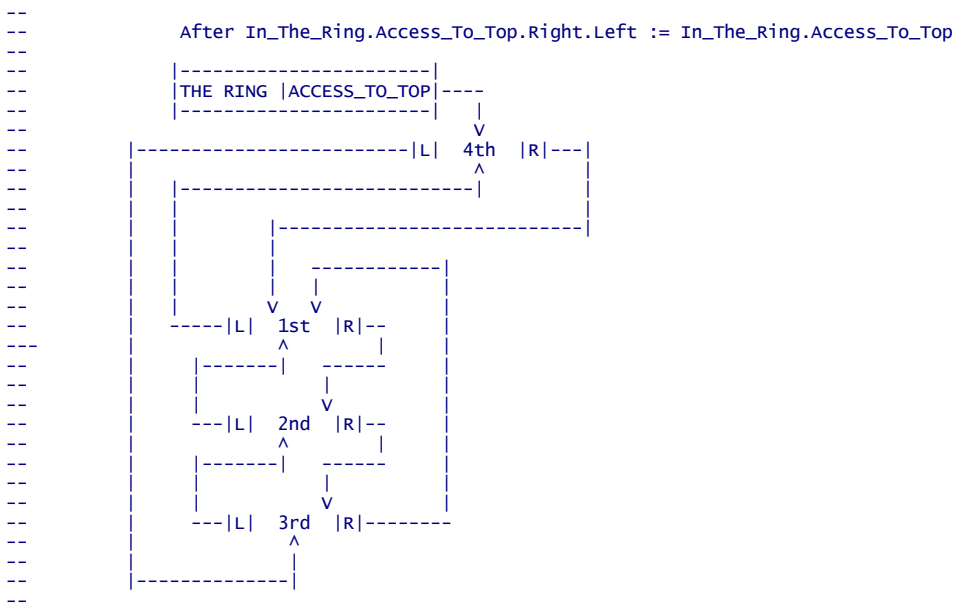
12.6.1.1 Ajout

Ici, le mécanisme d'addition est différent pour le premier élément de celui pour l'élément numéro n:

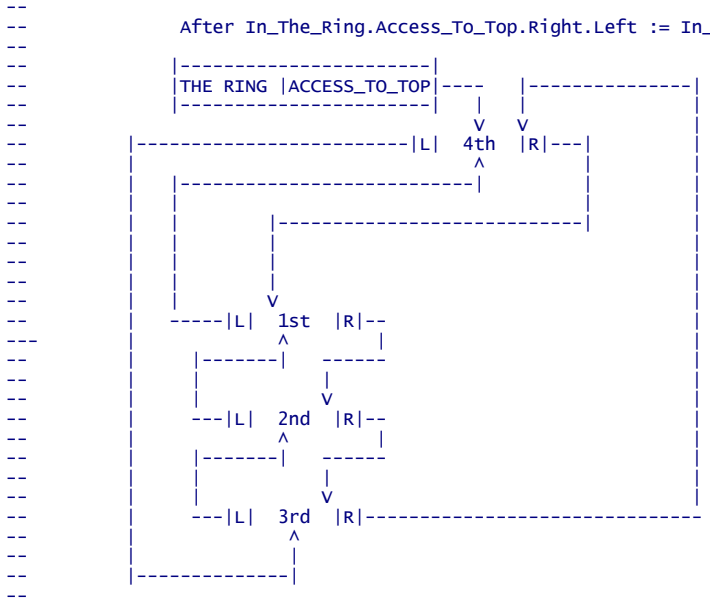
```
   procedure Insert_At_The_Left_Of_The_Top(The_Item    : in     Data_Type;
                                           In_The_Ring : in out Ring_Type)
      is
      begin
      --|
      --|  first time ?
      --|
      if In_The_Ring.Access_To_Top = null
         then
         --|
         --|create and attach node
         --|
         In_The_Ring.Access_To_Top := new Node_Type'(Left  => null,
                                                     Data  => The_Item,
                                                     Right => null);
         --|
         --| loop on itself
         --|
         In_The_Ring.Access_To_Top.Left  := In_The_Ring.Access_To_Top;
         In_The_Ring.Access_To_Top.Right := In_The_Ring.Access_To_Top;

         --|
         --|  Nth  time ?
         --|

         --                  S T A R T I N G   A T
         --
         --
         --             |-----------------------|
         --             |THE RING |ACCESS_TO_TOP|---|
         --             |-----------------------|   |
         --                                         |
         --                      |------------------|
         --                      |
         --                      |   ------------|
         --                      |   |           |
         --                      V   V           |
         --         |--------|L|  1st  |R|--     |
         ---        |          ^        |     |
         --         |     |-------|   ------     |
         --         |     |           |          |
         --         |     |           V          |
         --         |     ---|L|  2nd  |R|--     |
         --         |          ^        |     |
         --         |     |-------|   ------     |
         --         |     |           |          |
         --         |     |           V          |
         --         |     ---|L|  3rd  |R|--------
         --         |          ^
         --         |          |
         --         |--------------|
         --
         else
         --|
         --|create and attach node
         --|
         In_The_Ring.Access_To_Top := new Node_Type'(Left  => In_The_Ring.Access_To_Top.Left,
                                                     Data  => The_Item,
                                                     Right => In_The_Ring.Access_To_Top);
         --
         --              After create and attach node
         --
         --             |-----------------------|
         --             |THE RING |ACCESS_TO_TOP|----
         --             |-----------------------|   |
         --                                         V
         --         |-------------------------|L|  4th  |R|---|
         --         |                                         |
         --         |                                         |
         --         |                                         |
         --         |            |----------------------------|
         --         |            |
         --         |            |   ------------|
         --         |            |   |           |
         --         |            V   V           |
```

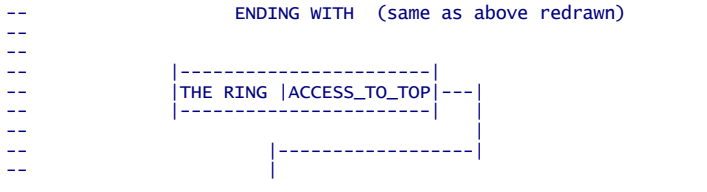


```
In_The_Ring.Access_To_Top.Right.Left := In_The_Ring.Access_To_Top;
```

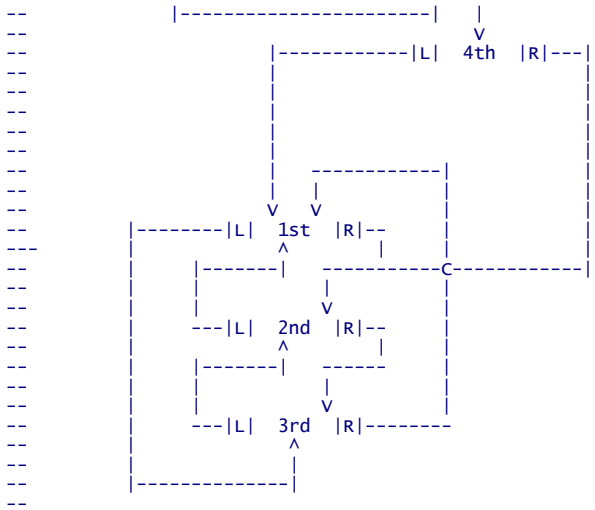


```
In_The_Ring.Access_To_Top.Left.Right := In_The_Ring.Access_To_Top;
```

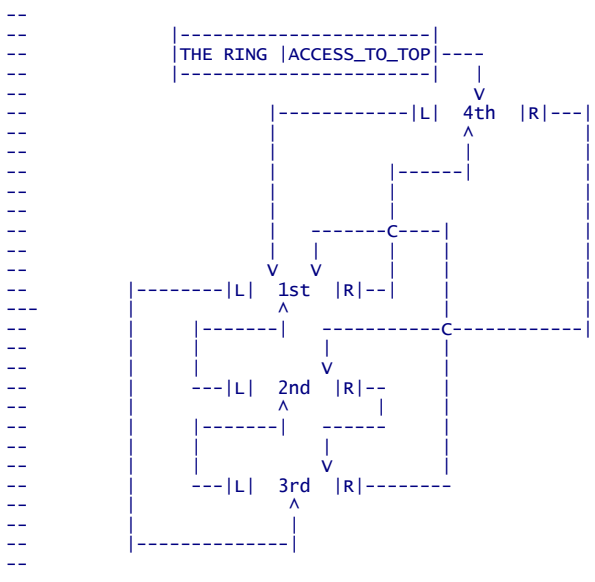




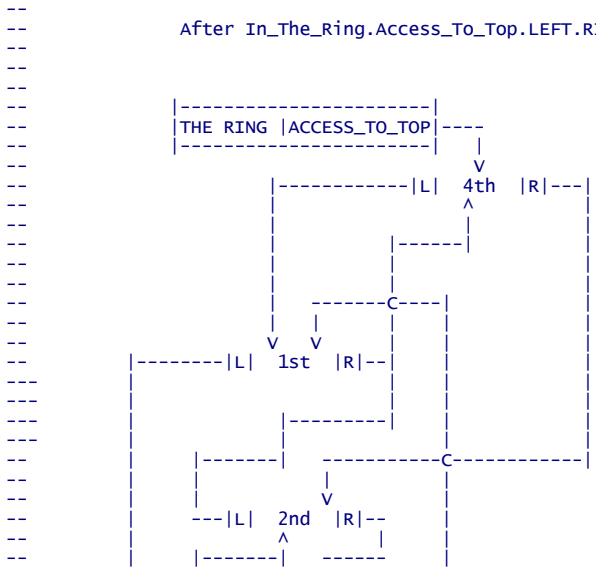
```
      --                     |   ------------|
      --                     |   |           |
      --                     V   V           |
      --          |--------|L|  4th  |R|--    |
      ---         |             ^        |    |
      --          |     |-------|   ------     |
      --          |     |           |          |
      --          |     |           V          |
      --          |     ---|L|  1st  |R|--     |
      --          |            ^        |     |
      --          |     |-------|   ------     |
      --          |     |           |          |
      --          |     |           V          |
      --          |     ---|L|  2nd  |R|--     |
      --          |            ^        |     |
      --          |     |-------|   ------     |
      --          |     |           |          |
      --          |     |           V          |
      --          |     ---|L|  3rd  |R|--------
      --          |             ^
      --          |             |
      --          |-------------|
      --
   end if;

   In_The_Ring.How_Many_Stored := 1  In_The_Ring.How_Many_Stored;

end Insert_At_The_Left_Of_The_Top;
--|
--|              |-------------------------------|
--|              | Insert_At_The_Left_Of_The_Top |
--|              |_______________________________|
--|
procedure Insert_At_The_Right_Of_The_Top(The_Item    : in     Data_Type;
                                         In_The_Ring : in out Ring_Type)
   is
   begin
   --|
   --|  first time ?
   --|
   if In_The_Ring.Access_To_Top = null
      then
      --|
      --|create and attach node
      --|
      In_The_Ring.Access_To_Top := new Node_Type'(Right => null,
                                                  Data  => The_Item,
                                                  Left  => null);
      --|
      --| loop on itself
      --|
      In_The_Ring.Access_To_Top.Right := In_The_Ring.Access_To_Top;
      In_The_Ring.Access_To_Top.Left  := In_The_Ring.Access_To_Top;

      --|
      --|  Nth  time ?
      --|

      --                  S T A R T I N G   A T
      --
      --
      --             |-----------------------|
      --             |THE RING |ACCESS_TO_TOP|---|
      --             |-----------------------|   |
      --                                        |
      --                      |-----------------|
      --                      |
      --                      |   ------------|
      --                      |   |           |
      --                      V   V           |
      --          |--------|L|  1st  |R|--     |
      ---         |             ^        |    |
      --          |     |-------|   ------     |
      --          |     |           |          |
      --          |     |           V          |
      --          |     ---|L|  2nd  |R|--     |
      --          |            ^        |     |
      --          |     |-------|   ------     |
      --          |     |           |          |
      --          |     |           V          |
      --          |     ---|L|  3rd  |R|--------
      --          |             ^
      --          |             |
      --          |-------------|
      --
      else
      --|
      --|create and attach node
      --|
      In_The_Ring.Access_To_Top := new Node_Type'(Left  => In_The_Ring.Access_To_Top,
                                                  Data  => The_Item,
                                                  Right => In_The_Ring.Access_To_Top.Right);
      --
      --              After create and attach node
      --
      --             |-----------------------|
      --             |THE RING |ACCESS_TO_TOP|----
```



```
--|
--| insert
--|
In_The_Ring.Access_To_Top.Left.Right := In_The_Ring.Access_To_Top;
```



```
In_The_Ring.Access_To_Top.Right.Left := In_The_Ring.Access_To_Top;
```



```
      --         |     |           |          |
      --         |     |           V          |
      --         |     ---|L|  3rd  |R|--------
      --         |              ^
      --         |              |
      --         |--------------|
      --

      --                   ENDING WITH  (same as above redrawn)
      --
      --
      --             |-----------------------|
      --             |THE RING |ACCESS_TO_TOP|---|
      --             |-----------------------|   |
      --                                         |
      --                      |------------------|
      --                      |
      --                      |   ------------|
      --                      |   |           |
      --                     V   V           |
      --         |--------|L|  4th  |R|--     |
      ---        |              ^       |     |
      --         |     |-------|   ------     |
      --         |     |           |          |
      --         |     |           V          |
      --         |     ---|L|  2nd  |R|--     |
      --         |              ^       |     |
      --         |     |-------|   ------     |
      --         |     |           |          |
      --         |     |           V          |
      --         |     ---|L|  3nd  |R|--     |
      --         |              ^       |     |
      --         |     |-------|   ------     |
      --         |     |           |          |
      --         |     |           V          |
      --         |     ---|L|  1st  |R|--------
      --         |              ^
      --         |              |
      --         |--------------|
      --
   end if;

   In_The_Ring.How_Many_Stored := 1  In_The_Ring.How_Many_Stored;


end Insert_At_The_Right_Of_The_Top;
--|
--|             |-------------------------------|
--|             |            Insert             |
--|             |_______________________________|
--|
procedure Insert(The_Item    : in     Data_Type;
                 In_The_Ring : in out Ring_Type;
                 Direction   : in     Direction_Type := Left_Sided)
   is
   begin

   if Direction = Left_Sided
      then
      Insert_At_The_Left_Of_The_Top(The_Item    => The_Item,
                                    In_The_Ring => In_The_Ring);
      elsif Direction = Right_Sided
      then

      Insert_At_The_Right_Of_The_Top(The_Item    => The_Item,
                                     In_The_Ring => In_The_Ring);
   end if;


end Insert;
```

12.6.1.2 Délétion

```
procedure Pop_To_Right(The_Ring : in out Ring_Type)
   is
   Z : Node_Access_Type := The_Ring.Access_To_Top;
   begin
   --|
   --|
   --|
   if The_Ring.Access_To_Top = The_Ring.Access_To_Top.Right
      then
      --|
      --|
      --|
      Free_Node(X => The_Ring.Access_To_Top);
      Free_Node(X => The_Ring.Access_To_Bookmark);
      else
      --|
      --|
      --|
```

```
      The_Ring.Access_To_Top.Left.Right := The_Ring.Access_To_Top.Right;
      The_Ring.Access_To_Top.Right.Left := The_Ring.Access_To_Top.Left;
      --|
      --|
      --|
      if The_Ring.Access_To_Bookmark = The_Ring.Access_To_Top
         then
         The_Ring.Access_To_Bookmark := null;
      end if;
      --|
      --|
      --|
      The_Ring.Access_To_Top := The_Ring.Access_To_Top.Right;
      Free_Node(X => Z);
   end if;


end Pop_To_Right;
--|
--|              |--------------|
--|              | Pop_To_Left  |
--|              |______________|
--|
procedure Pop_To_Left(The_Ring : in out Ring_Type)
   is
   Z : Node_Access_Type := The_Ring.Access_To_Top;
   begin
   --|
   --|
   --|
   if The_Ring.Access_To_Top = The_Ring.Access_To_Top.Left
      then
      --|
      --|
      --|
      Free_Node(X => The_Ring.Access_To_Top);
      Free_Node(X => The_Ring.Access_To_Bookmark);
      else
      --|
      --|
      --|
      The_Ring.Access_To_Top.Right.Left := The_Ring.Access_To_Top.Left;
      The_Ring.Access_To_Top.Left.Right := The_Ring.Access_To_Top.Right;
      --|
      --|
      --|
      if The_Ring.Access_To_Bookmark = The_Ring.Access_To_Top
         then
         The_Ring.Access_To_Bookmark := null;
      end if;
      --|
      --|
      --|
      The_Ring.Access_To_Top := The_Ring.Access_To_Top.Left;
      Free_Node(X => Z);
   end if;

end Pop_To_Left;
--|
--|              |--------------------------------|
--|              |            Delete_Top          |
--|              |________________________________|
--|
procedure Delete_Top(In_The_Ring : in out Ring_Type;
                     Direction   : in     Direction_Type := Left_Sided)
   is
   begin

   if Direction = Left_Sided
      then
      Pop_To_Left(The_Ring => In_The_Ring);
      elsif Direction = Right_Sided
      then
      Pop_To_Right(The_Ring => In_The_Ring);
   end if;

   In_The_Ring.How_Many_Stored := In_The_Ring.How_Many_Stored - 1;
end Delete_Top;
```

12.6.1.3 Valeur

```
function Top_Of(The_Ring : in     Ring_Type)
   return Data_Type
   is
   begin

   if null = The_Ring.Access_To_Top
      then
      raise Ring_Empty;
      else
      return The_Ring.Access_To_Top.all.Data;
   end if;
```

```
end Top_Of;
```

12.6.1.4 Initialiser

```
procedure Clear(The_Ring : in out Ring_Type)
   is
   begin

   while not (0 = The_Ring.How_Many_Stored) loop

      Delete_Top(In_The_Ring => The_Ring);

   end loop;

end Clear;
```

12.6.1.5 Rotation

```
procedure Rotate(The_Ring  : in out Ring_Type;
                 Direction : in     Direction_Type := Left_Sided)
   is
   begin

   if Direction = Left_Sided
      then
      The_Ring.Access_To_Top := The_Ring.Access_To_Top.Left;
      elsif Direction = Right_Sided
      then
      The_Ring.Access_To_Top := The_Ring.Access_To_Top.Right;
   end if;
end Rotate;
```

12.6.1.6 Itérateur

12.6.1.6.1 Initialiser

```
procedure Initialize(Iterator     :    out Iterator_Type;
                     For_The_Ring : in out Ring_Type;
                     Direction    : in     Direction_Type := Left_Sided)
   is
   begin
   Iterator := (Current_Node_Access => For_The_Ring.Access_To_Top,
                Count               => 0,
                Count_Until         => For_The_Ring.How_Many_Stored,
                Side                => Direction);

   To_Next_Value(Iterator => Iterator);
end Initialize;
```

12.6.1.6.2 Valeur

```
function Current_Value(Iterator : in     Iterator_Type)
   return Data_Type
   is
   begin
   return Iterator.Current_Node_Access.Data;
end Current_Value;
```

12.6.1.6.3 Suivant

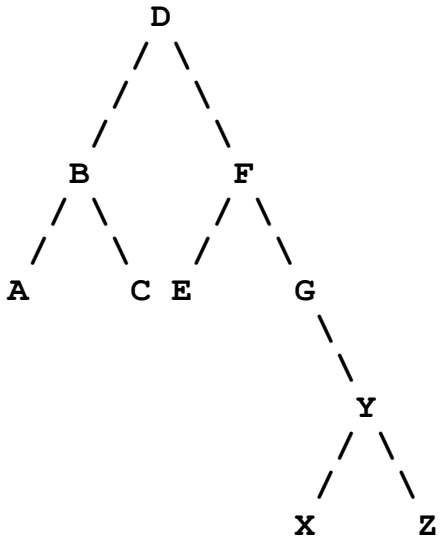
```
procedure To_Next_Value(Iterator : in out Iterator_Type)
   is
   begin
   if Iterator.Side = Left_Sided
      then
      Iterator.Current_Node_Access := Iterator.Current_Node_Access.Left;
      else
      Iterator.Current_Node_Access := Iterator.Current_Node_Access.Right;
   end if;
   Iterator.Count := 1  Iterator.Count;
end To_Next_Value;
```

12.6.1.6.4 Fini

```
function Is_Done(Iterator : in     Iterator_Type)
   return Boolean
   is
   begin
   return Iterator.Count = 1  Iterator.Count_Until;
end Is_Done;
```

12.7 L'ARBRE BINAIRE

Voici un exemple d'arbre binaire:



Un arbre binaire est une structure ordonnée. Les éléments sont stockés dans les noeuds de l'arbre (B, D, F, G, Y). Les noeuds externes de l'arbre sont appelés feuilles (A, C, E, X, Z). Dans l'exemple ci-dessus les éléments stockés sont simples sont des lettres pour lesquels la clé et l'élément contenu sont identiques. Les opérateurs =, > et < sont définis par l'ordre alphabétique de la lettre qui est à la fois le contenu et la clé. Chaque élément a un parent (sauf celui du haut (D) qui est le sommet) et peut avoir un fils gauche et/ou un fils droit.

La relation est :
clé de l'élément du fils gauche < clé de l'élément du parent < clé de l'élément du fils droit.

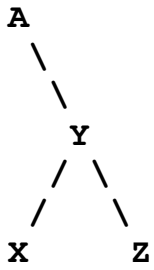
Elle est respectée dans tout l'arbre et a comme conséquence que le stockage deux deux élément dont la clé est identique est impossible. En général, il est utile de distinguer la clé de l'élément dans le cas d'éléments composites : par exemple numéro de compte en banque, nom du titulaire, adresse du titulaire,....

Les opérations usuelles : initialisation, ajout, suppression, recherche sont bien entendu définis. Par contre, la notion d'ordre d'introduction n'existe pas **M A I S** l'ordre d'ajout des éléments détermine la forme de l'arbre.

Une nouvelle opération est définie: **le parcours d'un arbre**. Il y a trois types de parcours classiques: pre-order, in-order, post-order. La base commune est le parcours d'un arbre qui commence au sommet et est défini comme suit pour chaque noeud:

Pour chaque noeud ou feuille de l'arbre :

1. Arrivée depuis le parent (sauf début = sommet).
2. Retour Immédiat vers le parent si le noeud est vide
3. Visite "pre-order"
4. Départ RÉCURSIF vers le fils gauche .
5. Retour du fils gauche.
6. Visite "in-order"
7. Départ RÉCURSIF vers le fils droit .
8. Retour du fils droit.
9. Visite "post-order"
10. Retour vers le parent (ou fin si sommet).



Le résultat des trois types de parcours sont:

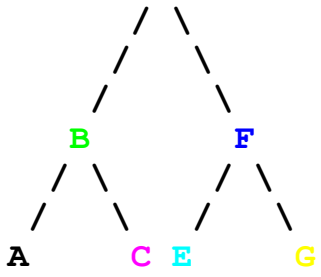
"Pre-order " : A Y X Z
"In-order " : A X Y Z
"Post-order" : X Z Y A

12.7.1 L'arbre binaire: Implémentation avec un tableau

L'implémentation d'un arbre binaire dans un tableau est illustrée ci-dessous:

```
|---------------|
| T A B L E A U |
|---------------|                    D
```

Indice	Contenu
1	D
2	B
3	F
4	A
5	C
6	E
7	G



Le calcul de l'indice du tableau est déterminé selon:

1. Le sommet à l'indice 1.
2. Le fils gauche a l'indice du parent * 2.
3. Le fils droit a l'indice du parent * 2 +1.

L'indice du parent peut être obtenu en divisant par 2 l'indice des fils.

Les différents algorithmes correspondant aux opérations nécessitent un élément "neutre" permettant d'identifier une case vide dans le tableau et donc un noeud/feuille qui n'existe pas (encore). La clé de comparaison étant spécifique de l'application, il est nécessaire de définir les opérateurs de comparaisons en prenant soin de renvoyer faux pour toute comparaison avec l'élément neutre !!! (voir l'implantation du générique).

La partie générique:

```
generic

  type Data_Type is private;
  Max_Number_Of_Items : Positive;
  Null_Data           : Data_Type;
La clé peut être différente de l'élément
  --|
  --| for = operator
  --|
  with function "=" (Left  : in Data_Type;
                     Right : in     Data_Type) return Boolean is <>;
  --|
  --| for > operator
  --|
  with function ">" (Left  : in Data_Type;
                     Right : in     Data_Type) return Boolean is <>;


  package Simple_Bin_Tree
    is

    Bin_Tree_Full              : exception;
    Bin_Tree_Empty             : exception;
    Record_Not_Found_On_Delete : exception;

    type Iterator_Type is limited private;
    type Bin_Tree_Type is limited private;
Nouveau
    type Iterator_Kind_Type is (Pre_Order, In_Order, Post_Order);
```

Limited private: pour **obliger l'utilisateur à utiliser les opérations du package et interdire l'assignation.**

```
    private
    --|
    --|  BIN_TREE table
    --|
    type Bin_Tree_Table_Array_Type is array (1 .. 2 ** Max_Number_Of_Items - 1) of Data_Type;

    type Bin_Tree_Type
       is
       record
       Bin_Tree_Table_Array : aliased Bin_Tree_Table_Array_Type;
       How_Many             : Natural := 0;
    end record;
    --|
    --|  needed for the iterator
    --|
    type Bin_Tree_Table_Array_Access_Type is access all Bin_Tree_Table_Array_Type;
    --|
    --|  iterator
    --|
    type Iterator_Type is
       record
       Iterator_Kind              : Iterator_Kind_Type              := In_Order;
       Current_Index              : Integer                         := 0;
       Previous_Index             : Integer                         := 0;
```

```
      Bin_Tree_Table_Array_Access : Bin_Tree_Table_Array_Access_Type := null;

   end record;
   Post_Order_Done : Boolean := False;
```

Example d'instantiation:

```
with Simple_Bin_Tree;
with Ada.Text_Io;
with Calendar;

procedure Test_Simple_Bin_Tree
  is
  --|
  --|  the BIN_TREE table
  --|
  Tree_Depth : constant := 19;
  --|
  --| the algo needs these   to be defined as not in normal use
  --|
  Null_Record : constant Integer := Integer'First;
  --|
  --| instantiation
  --|
  package My_Bin_Tree is new Simple_Bin_Tree(Data_Type           => Integer,
                                             Null_Data           => Null_Record,
                                             Max_Number_Of_Items => Tree_Depth,
                                             "="                 => "=",
                                             ">"                 => ">");
  --|
  --| general data
  --|
  A_Bin_Tree              : My_Bin_Tree.Bin_Tree_Type;
  My_Iterator             : My_Bin_Tree.Iterator_Type;
```

Example d'utilisation:

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

      My_Bin_Tree.Add(Data            => I,
                      In_The_Bin_Tree => A_Bin_Tree);


      My_Bin_Tree.Seek(Data            => I,
                       In_The_Bin_Tree => A_Bin_Tree,
                       Success         => Ok,


   My_Bin_Tree.Clear(The_Bin_Tree => A_Bin_Tree);

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

PRE ORDER ITERATOR

```
   My_Bin_Tree.Initialize(Iterator         => My_Iterator,
                          For_The_Bin_Tree => A_Bin_Tree,
                          Iterator_Kind    => My_Bin_Tree.Pre_Order);
   --|
   --| print all string in BIN_TREE table
   --|
   while not My_Bin_Tree.Is_Done(Iterator => My_Iterator) loop

      My_Data := My_Bin_Tree.Current_Value(Iterator => My_Iterator);

      Ada.Text_Io.Put(Integer'Image(My_Data));

      My_Bin_Tree.To_Next_Value(Iterator => My_Iterator);
   end loop;
```

IN ORDER ITERATOR

```
   My_Bin_Tree.Initialize(Iterator         => My_Iterator,
                          For_The_Bin_Tree => A_Bin_Tree,
                          Iterator_Kind    => My_Bin_Tree.In_Order);

   while not My_Bin_Tree.Is_Done(Iterator => My_Iterator) loop

      My_Data := My_Bin_Tree.Current_Value(Iterator => My_Iterator);

      Ada.Text_Io.Put(Integer'Image(My_Data));

      My_Bin_Tree.To_Next_Value(Iterator => My_Iterator);
   end loop;
```

POST ORDER ITERATOR

```
   My_Bin_Tree.Initialize(Iterator         => My_Iterator,
                          For_The_Bin_Tree => A_Bin_Tree,
                          Iterator_Kind    => My_Bin_Tree.Post_Order);
```

```
while not My_Bin_Tree.Is_Done(Iterator => My_Iterator) loop

   My_Data := My_Bin_Tree.Current_Value(Iterator => My_Iterator);

   Ada.Text_Io.Put(Integer'Image(My_Data));

   My_Bin_Tree.To_Next_Value(Iterator => My_Iterator);
end loop;
```

12.7.1.1 Données et Types de données:

```
type Bin_Tree_Table_Array_Type is array (1 .. 2 ** Max_Number_Of_Items - 1) of Data_Type;

type Bin_Tree_Type
   is
   record
   Bin_Tree_Table_Array : aliased Bin_Tree_Table_Array_Type;
   How_Many             : Natural := 0;
end record;

type Bin_Tree_Table_Array_Access_Type is access all Bin_Tree_Table_Array_Type;

type Iterator_Type is
   record
   Iterator_Kind              : Iterator_Kind_Type              := In_Order;
   Current_Index              : Integer                         := 0;
   Previous_Index             : Integer                         := 0;
   Bin_Tree_Table_Array_Access : Bin_Tree_Table_Array_Access_Type := null;
end record;
```

12.7.1.2 Ajout

Le but est de trouver un emplacement vide pour mettre le nouvel élément. Commencer au sommet en mettant l'indice à 1
index←1
Boucle tant que le noeud/feuille n'est pas vide (non égal)
tant que tableau(index) ≠ vide boucle
Si la nouvelle clé est < que la clé du noeud courant
si tableau(index) < clé
index←index *2
Aller au fils gauche
sinon
Aller au fils droit
index←index *2 +1
fin si
fin boucle
Insérer le nouvel élément
tableau(index) ← élément

```
procedure Add(Data            : in     Data_Type;
              In_The_Bin_Tree : in out Bin_Tree_Type)
      is
      Index : Positive := 1;
      begin

      while not (Null_Data = In_The_Bin_Tree.Bin_Tree_Table_Array(Index)) loop

         if Data > In_The_Bin_Tree.Bin_Tree_Table_Array(Index)
            then
            Index := 2 * Index  1;
            else
            Index := 2 * Index;
         end if;

      end loop;

      In_The_Bin_Tree.Bin_Tree_Table_Array(Index) := Data;
      In_The_Bin_Tree.How_Many                    := In_The_Bin_Tree.How_Many  1;
Tableau Plein !!!!
      exception
      when Constraint_Error =>
      Ada.Text_Io.Put("add" & Integer'Image(Index) & Integer'Image(In_The_Bin_Tree.How_Many));
   end Add;
```

12.7.1.3 Délétion

La délétion d'un élément est l'opération la plus compliquée. Après avoir trouvé dans l'arbre l'élément à supprimer appelé A, il y a trois cas à prendre en compte: pas de fils, un seul fils ou deux fils:

1. L'élément A à supprimer est stocké dans une feuille, elle n'a donc pas de fils et sa suppression est simple: le contenu est remplacé par l'élément «vide».
2. L'élément A à supprimer est stocké dans un noeud qui n'a qu'un seul fils: on le remplace A par le sous arbre dont le sommet est le fils.
3. L'élément A à supprimer est stocké dans un noeud qui a deux fils:
   La première étape consiste a trouver:

- Ou bien son successeur in-order (appelons le B) qui est le fils le plus à gauche du sous arbre droit de A
- Ou bien son prédécesseur in-order (appelons le B) qui est le fils le plus à droite du sous arbre gauche de A.

   Comme aucun de ces noeuds ne peut avoir plus d'un fils sinon ils ne peuvent pas être le prédécesseur ou le successeur "in-order", il peut être supprimé en utilisant le cas numéro 2.

4. Une fois B trouvé par une des méthodes ci-dessus, on remplace A par le sous arbre dont B est le sommet et on supprime cet élément avec deux cas : il n'a pas de fils (cas numéro 1) ou il a un fils cas numéro 2).

L'opération "*remplace A par le sous arbre*" apparaît deux fois et sera donc implémentée séparément pour ne pas dupliquer un code qui est complexe et sera étudié à part.

12.7.1.3.1 La recherche de l'élément à supprimer

```
Commencer au sommet en mettant l'indice à 1
index←1
Boucle tant que le noeud/feuille n'est pas vide (non égal)
tant que tableau(index)   ≠  vide  boucle
Si la nouvelle clé est < que la clé du noeud courant
si tableau(index) < clé
index←index * 2
fin si
Aller au fils gauche
si tableau(index) > clé
Aller au fils droit
index←index * 2 +1
sinon trouvé: appel suppression
fin si
fin boucle
Non trouvé erreur !!!
```

12.7.1.3.2 Suppression: le choix entre les trois cas

```
Test aucun fils
si tableau(Indice * 2 + 1) = vide
et  tableau(Indice * 2) = vide ...
Test deux fils
si tableau(Indice * 2 + 1) ≠ vide
et  tableau(Indice * 2) ≠ vide
Sinon un seul fils
sinon ...
```

12.7.1.3.3 Le cas où il n'y a aucun fils

```
Mettre l'élément vide dans cette case du tableau.
tableau(index) ← élément neutre
```

12.7.1.3.4 Le cas où il n'y a qu'un seul fils

Appel de Remplacement par le sous arbre pour le noeud de l'élément trouvé
remplacer_Parent(Indice)

12.7.1.3.5 Le cas où il y a deux fils.

Trouver l'indice du successeur in-order qui est le fils le plus à gauche du sous arbre droit de indice de l'élément à supprimer
Aller au fils droit
index←index * 2 +1
trouver le fils le plus à gauche qui n'a pas de fils gauche
Tant que tableau(index* 2) ≠ vide boucle
index←index * 2 +1
fin boucle
échanger les 2
tableau(indice de l'élément à supprimer) ← tableau(index)
Il reste deux cas: existence ou non d'un fils droit (il ne peut pas y avoir de fils gauche ou ce n'est pas le successeur in-order):
Pas de fils?
si tableau(Indice * 2 + 1) = vide
tableau(Indice * 2) ← vide
Un fils droit: attacher le sous arbre au parent **INDICE / 2**
sinon
remplacer_Parent(Indice/2)
fin si

12.7.1.3.6 Remplacement par le sous arbre

Les éléments étant stockés dans un tableau, la solution correspond à une modification des indices des éléments concernés:

1. Identifier et stocker les indices concernés dans un tableau. Le sous arbre correspondant sera visité "in-order" et les indices conservés dans le tableau.
2. Calculer et stocker les corrections à appliquer.
3. Traiter le cas du premier indice en le divisant par 2.
4. Traiter les autres indices s'ils existent en utilisant les corrections à appliquer.

Identifier et stocker les indices concernés dans un tableau.

Le sous arbre correspondant sera visité "pre-order" et les indices conservés dans le tableau.

La nature récursive du parcours impose:
i←1
Fill_Table(Index)
si tableau(Indice) = vide
retour
fin si
table(i)← index
i← i+1
Fill_Table(2 * Index)
Fill_Table(2 * Index +1)

Calculer et stocker les corrections à appliquer.
Un arbre binaire ayant une construction logarithmique base 2, la table de correction suit la même logique. Voici ce qu'il faut réaliser: pour chaque index il faut soustraire la correction notée corr. Dans la table ci-dessous. Par exemple, l'élément stocké dans le tableau à l'indice 45 devra être déplacé à l'indice 29= 45 – 16. La création de ce tableau se fait comme suit:

Dummy ← 6
Sub ← 2
boucle de 3 à la dernière valeur de l'indice
si Dummy=Indice de la boucle
Sub ← 2 * Sub
Dummy ← 2 * Dummy
fin si
tableau de correction(i) ← Sub

fin boucle

Index	Corr.	Index	Corr.	Index	Corr.	Index	Corr.	Index	Corr.	Index	Corr.	Index	Corr.	Index	Corr.
3	2	4	2	5	2	6	4	7	4	8	4	9	4	10	4
11	4	12	8	13	8	14	8	15	8	16	8	17	8	18	8
19	8	20	8	21	8	22	8	23	8	24	16	25	16	26	16
27	16	28	16	29	16	30	16	31	16	32	16	33	16	34	16
35	16	36	16	37	16	38	16	39	16	40	16	41	16	42	16
43	16	44	16	45	16	46	16	47	16	48	32	49	32	50	32
51	32	52	32	53	32	54	32	55	32	56	32	57	32	58	32
59	32	60	32	61	32	62	32	63	32	64	32	65	32	66	32
67	32	68	32	69	32	70	32	71	32	72	32	73	32	74	32
75	32	76	32	77	32	78	32	79	32	80	32	81	32	82	32
83	32	84	32	85	32	86	32	87	32	88	32	89	32	90	32
91	32	92	32	93	32	94	32	95	32	96	64	97	64	98	64
99	64	100	64	101	64	102	64	103	64	104	64	105	64	106	64
107	64	108	64	109	64	110	64	111	64	112	64	113	64	114	64
115	64	116	64	117	64	118	64	119	64	120	64	121	64	122	64
123	64	124	64	125	64	126	64	127	64	128	64	129	64	130	64
131	64	132	64	133	64	134	64	135	64	136	64	137	64	138	64
139	64	140	64	141	64	142	64	143	64	144	64	145	64	146	64
147	64	148	64	149	64	150	64	151	64	152	64	153	64	154	64
155	64	156	64	157	64	158	64	159	64	160	64	161	64	162	64
163	64	164	64	165	64	166	64	167	64	168	64	169	64	170	64
171	64	172	64	173	64	174	64	175	64	176	64	177	64	178	64
179	64	180	64	181	64	182	64	183	64	184	64	185	64	186	64
187	64	188	64	189	64	190	64	191	64	192	128	193	128	.....	.....

Traiter le cas du premier indice en le divisant par 2.

Parent devient fils en utilisant table(2) qui contient le fils
tableau(table(2)/2) ← tableau(table(2))
Fils devient vide
tableau(table(2)) ← vide

Traiter les autres indices s'ils existent en utilisant les corrections à appliquer.

Boucle de 3 au dernier élément dans le sous arbre
Tableau(table(i) - tableau de correction(i)) ← tableau(table(i))
Tableau(table(i)) ← vide
fin boucle

Le code complet :

```
procedure Remove(The_Bin_Tree : in out Bin_Tree_Type;
                 Data         : in     Data_Type)
   is

   --
   --                There are several cases to consider:
   --
   --1) Deleting a leaf: Deleting a node with no children is easy,
   --   as we can simply remove it from the tree.
   --
   --2) Deleting a node with one child: Delete it and replace it with its child.
   --
   --3) Deleting a node with two children:
   --   Suppose the node to be deleted is called Z.
   --   We replace the value of Z with either its in-order successor
   --   (the left-most child of the right subtree)
   --   or the in-order predecessor (the right-most child of the left subtree).
```

```
--   Once we find either the in-order successor or predecessor,
--   swap it with Z, and then delete it.
--   Since either of these nodes must have less than two children
--   (otherwise it cannot be the in-order successor or predecessor),
--   it can be deleted using the previous two cases.
--
--  In a good implementation, it is generally recommended
--  to avoid consistently using one of these nodes,
--  because this can unbalance the tree  (to be done).

--|                      |_____________________________________|
--|                      |                                     |
--|                      |      Delete_A_Node                  |
--|                      |_____________________________________|
--|                      |                                     |
--|
procedure Delete_A_Node(Index_Of_Node_To_Be_Deleted : in     Positive)
   is

   type Table_Type is array (1 .. 2 ** Max_Number_Of_Items - 1) of Natural;
   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |   Attach_Subtree_To_Parent          |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   procedure Attach_Subtree_To_Parent(Index_Of_Current : in     Positive)
      is
      Index                     : Natural    := Index_Of_Current;
      Table_Index               : Positive   := Positive'First;
      Table                     : Table_Type := (others => 0);

      procedure Fill_Table(Index : in     Positive)
         is
         begin
         --|
         --| end of branch  ?    go back up
         --|
         if Null_Data = The_Bin_Tree.Bin_Tree_Table_Array(Index)
            then
            return;
         end if;
         --|
         --| LOAD INDEX PRE ORDER !!!!!
         --|
         Table(Table_Index) := Index;
         Table_Index        := 1 + Table_Index;
         --|
         --|  none of the above : keep walking the tree (customary dual recursion)
         --|
         Fill_Table(Index => 2 * Index);
         Fill_Table(Index => 2 * Index + 1);

      end Fill_Table;
      Subtract_Table : Table_Type := (others => 0);
      Dummy_Index    : Positive   := Positive'First;
      Sub_Value      : Positive   := Positive'First;

      begin
      --|
      --|  create a table to change the indexes in the tree by substraction
      --|
      Dummy_Index := 6;
      Sub_Value   := 2;

      for I in 3 .. Subtract_Table'Last loop
         if I = Dummy_Index
            then
            Sub_Value   := 2 * Sub_Value;
            Dummy_Index := 2 * Dummy_Index;
         end if;
         Subtract_Table(I) := Sub_Value;
      end loop;

      Table := (others => 0);
      Fill_Table(Index => Index_Of_Current);


      --|
      --|  care for the first one :: go up one step :=>  /2
      --|
      The_Bin_Tree.Bin_Tree_Table_Array(Table(2) / 2) :=
      The_Bin_Tree.Bin_Tree_Table_Array(Table(2));
      The_Bin_Tree.Bin_Tree_Table_Array(Table(2)) := Null_Data;
      --|
      --|  care for the others using substraction table
      --|
      for I in 3 .. Table_Index - 1 loop
         if Table(I) /= 0
            then
            The_Bin_Tree.Bin_Tree_Table_Array(Table(I) - Subtract_Table(Table(I))) :=
            The_Bin_Tree.Bin_Tree_Table_Array(Table(I));
            The_Bin_Tree.Bin_Tree_Table_Array(Table(I)) := Null_Data;
         end if;
      end loop;
```

```
end Attach_Subtree_To_Parent;

Index_Of_Successor : Positive;
Index_Of_Current   : Positive := Index_Of_Node_To_Be_Deleted;

begin

--|
--|=============================================
--| The Node to be deleted has no children
--|=============================================
--|
if The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2 + 1) = Null_Data
   and then The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2) = Null_Data
   then
   The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current) := Null_Data;
   --|
   --|=======================================================
   --| The Node To Be Deleted Has A Right Child But No Left Child
   --|=======================================================
   --|
   elsif The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2 + 1) /= Null_Data
   and then The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2) = Null_Data
   then
   --|
   --|Attach right subtree To Parent
   --|
   Attach_Subtree_To_Parent(Index_Of_Current => Index_Of_Current);
   --|
   --|=======================================================
   --| The Node To Be Deleted Has A Left Child But No Right Child
   --|=======================================================
   --|
   elsif The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2 + 1) = Null_Data
   and then The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2) /= Null_Data
   then
   --|
   --|Attach left subtree To Parent
   --|
   Attach_Subtree_To_Parent(Index_Of_Current => Index_Of_Current);
   --|
   --|=============================================
   --| The Node To Be Deleted Has Two Children
   --|=============================================
   --|
   elsif The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2 + 1) /= Null_Data
   and then The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2) /= Null_Data
   then



   --3) Deleting a node with two children:
   --   Suppose the node to be deleted is called Z.
   --   We replace the value of Z with either its in-order successor
   --   (the left-most child of the right subtree)
   --   or the in-order predecessor (the right-most child of the left subtree).
   --   Once we find either the in-order successor or predecessor,
   --   swap it with Z, and then delete it.
   --   Since either of these nodes must have less than two children
   --   (otherwise it cannot be the in-order successor or predecessor),
   --   it can be deleted using the previous two cases.
   --
   --  In a good implementation, it is generally recommended
   --  to avoid consistently using one of these nodes,
   --  because this can unbalance the tree  (to be done).

   --|
   --| get inorder successor  :  the left-most child of the right subtree
   --|

   --|
   --| 1) get to the right subtree
   --|
   Index_Of_Current := 2 * Index_Of_Current + 1;
   --|
   --| 2) find out the leftmost child: it has no left child !
   --|
   while not (The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current) /= Null_Data) loop
      Index_Of_Current := 2 * Index_Of_Current;
   end loop;
   --|
   --| swap  both
   --|
   The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Node_To_Be_Deleted)
   := The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current);
   --|
   --| two cases left: no child at all or a right child
   --|
   if The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2 + 1) = Null_Data
      and then The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2) = Null_Data
      then
      --|
      --|no child clear node
      --|
      The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current) := Null_Data;
      else
```

```
            --|
            --|a righ child : Attach right subtree To  PARENT =====>>> (Index_Of_Current/2)
            --|
            Attach_Subtree_To_Parent(Index_Of_Current => Index_Of_Current / 2);
         end if;
      end if;
   end Delete_A_Node;

   Found : Boolean  := False;
   Index : Positive := 1;
   --|
   --| find the node to be removed
   --|
   begin

   while not (Null_Data = The_Bin_Tree.Bin_Tree_Table_Array(Index)) loop

      if Data > The_Bin_Tree.Bin_Tree_Table_Array(Index)
         then
         Index := 2 * Index + 1;
         elsif Data = The_Bin_Tree.Bin_Tree_Table_Array(Index)
         then
         Found := True;
         exit;
         else
         Index := 2 * Index;
      end if;
   end loop;
   --|
   --| ok remove else raise error
   --|
   if Found
      then
      Delete_A_Node(Index_Of_Node_To_Be_Deleted => Index);
      else
      raise Record_Not_Found_On_Delete;
   end if;
   --|
   --| keep track
   --|
   The_Bin_Tree.How_Many := The_Bin_Tree.How_Many - 1;

end Remove;
```

12.7.1.4 Recherche

Commencer au sommet en mettant l'indice à 1
index←1
Boucle tant que le noeud/feuille n'est pas vide (non égal)
tant que tableau(index) ≠ vide boucle
Si la nouvelle clé est < que la clé du noeud courant
si tableau(index) < clé
index←index * 2
fin si
Aller au fils gauche
si tableau(index) > clé
Aller au fils droit
index←index * 2 +1
sinon trouvé
fin si
fin boucle
Non trouvé

Code:

```
procedure Seek(In_The_Bin_Tree : in     Bin_Tree_Type;
               Data            : in     Data_Type;
               Success         :    out Boolean;
               Found_Record    :    out Data_Type)
   is

   Index : Positive := 1;
   begin

   while not (Null_Data = In_The_Bin_Tree.Bin_Tree_Table_Array(Index)) loop

      if Data > In_The_Bin_Tree.Bin_Tree_Table_Array(Index)
         then
         Index := 2 * Index  1;

         elsif Data = In_The_Bin_Tree.Bin_Tree_Table_Array(Index)
         then
         Success      := True;
         Found_Record := In_The_Bin_Tree.Bin_Tree_Table_Array(Index);
         return;
         else
```

```
         Index := 2 * Index;
      end if;

   end loop;

   Success := False;
   exception
   when Constraint_Error =>
   Ada.Text_Io.Put(Integer'Image(Index));
   Success := False;
end Seek;
```

12.7.1.5 Initialiser

Le tableau est initialisé utilisant "l'élément neutre" et lenombre d'éléments est mis à zéro.

```
procedure Clear(The_Bin_Tree : in out Bin_Tree_Type)
   is
   begin
   The_Bin_Tree.Bin_Tree_Table_Array := (others => Null_Data);
   The_Bin_Tree.How_Many             := 0;
end Clear;
```

12.7.1.6 itérateur (pre-order, in-order & post-order)

Les itérateurs sont basés sur le parcours de l'arbre binaire. Ici une difficulté supplémentaire: Le parcours classique par double récursion n'est pas utilisable. En effet, il est impossible dans un itérateur "actif" de sortir puis de rentrer au beau milieu d'une récursion. La solution itérative est donc seule possible. Dans ce cas, il est possible d'effectuer les *"tours de boucle"* un par un.
Considérons chaque noeud (ou feuille) de l'arbre:
pour différencier les 3 origines possibles du retour à ce noeud il faut utiliser 2 variables: l'indice courant dans le tableau, e***t l'indice précédent***.

A chaque étape du parcours itératif, les conditions suivantes peuvent se présenter:

> L'élément contenu dans le noeud visité est "l'élément neutre": la position est vide => retour immédiat vers le parent en divisant l'indice par 2. Sinon les trois choix suivants sont appliqués:

1. C'est la première visite car l'indice courant / 2 = l'indice précédent : départ pour le sous arbre gauche.
2. C'est la seconde visite (retour du sous arbre gauche) car l'indice courant * 2 = l'indice précédent : départ pour le sous arbre droit.
3. C'est la troisième visite (retour du sous arbre droit) car l'indice courant * 2 +1 = l'indice précédent : retour au noeud parent.

Les détails: ils sont décrits ci-dessous.

12.7.1.6.1 Initialiser

Les variables de l'itérateur sont initialisées (noter en particulier le pointeur vers le tableau de données). La procédure to_next_value est appelée pour que la fonction valeur renvoie le premier élément s'il existe. S'il n'existe pas, la fonction fini sera vrai.

```
procedure Initialize(Iterator         :    out Iterator_Type;
                     For_The_Bin_Tree : in     Bin_Tree_Type;
                     Iterator_Kind    : in     Iterator_Kind_Type := In_Order)
   is

   begin
   Iterator := (Iterator_Kind              => Iterator_Kind,
                Previous_Index             => 0,
                Current_Index              => 1,
                Bin_Tree_Table_Array_Access =>
                new Bin_Tree_Table_Array_Type'(For_The_Bin_Tree.Bin_Tree_Table_Array));
   --|
   --| get first value if any
   --|
   To_Next_Value(Iterator => Iterator);

end Initialize;
```

12.7.1.6.2 Valeur

Cette fonction est appelée **APRÈS** le passage au suivant donc l'indice à utiliser est l'indice précédent.

```
function Current_Value(Iterator : in     Iterator_Type) return Data_Type
       is
       Data : Data_Type;
       begin
       return Iterator.Bin_Tree_Table_Array_Access.all (Iterator.Previous_Index);
    end Current_Value;
```

12.7.1.6.3 Suivant

Cette procédure utilise le parcours décrit plus haut et sort de la boucle en trois positions correspondant and trois types d'itération implémentés.

Boucle tant que la parcours n'est pas terminé.
Boucle tant que non vrai (indice=0 et précédent =1)
Vide ? retour au parent, et continuation de la boucle
Si table(index)=élément neutre
précédent← index
index ← index /2

Première fois ??
ou alors si index/2= précédent
Si oui, preparation de l'indice pour un départ vers le sous arbre gauche qui ne sera effectif que durant l'appel suivant à la procédure , puis sortie de procédure si le parcours est pre-order, sinon la boucle principale continue.
précédent← index
index ← index * 2
Sortie de procédure (ou de boucle principale) si pre-order
si pre_order
return (exit)
fin si
Seconde fois ??
ou alors si index*2= précédent
Si oui, preparation de l'indice pour un départ vers le sous arbre droit qui ne sera effectif que durant l'appel suivant à la procédure , puis sortie de procédure si le parcours est pre-order, sinon la boucle principale continue.
précédent← index
index ← index * 2 +1
Sortie de procédure (ou de boucle principale) si in-order
si in_order
return (exit)
fin si
Troisième fois ??
ou alors si index*2 +1= précédent
Si oui, preparation de l'indice pour un retour vers le parent qui ne sera effectif que durant l'appel suivant à la procédure , puis sortie de procédure si le parcours est pre-order, sinon la boucle principale continue.
précédent← index
index ← index /2
Sortie de procédure (ou de boucle principale) si post-order
si post_order
return (exit)
Fin si
Fin si
Fin boucle.

Le code:

```
     procedure To_Next_Value(Iterator : in out Iterator_Type)
       is
       begin

       while not (Iterator.Current_Index = 0 and then Iterator.Previous_Index = 1) loop
          --|
          --|1) the position is void : go up by dividing by 2
          --|
          if Null_Data = Iterator.Bin_Tree_Table_Array_Access.all (Iterator.Current_Index)
             then
```

```
            Iterator.Previous_Index := Iterator.Current_Index;
            Iterator.Current_Index  := Iterator.Previous_Index / 2;
            --|
            --|2) it is the first time we arrive in this node (current/2 = previous)   go left
            --|
            elsif Iterator.Current_Index / 2 = Iterator.Previous_Index
            then
            Iterator.Previous_Index := Iterator.Current_Index;
            Iterator.Current_Index  := Iterator.Previous_Index * 2;

            if Pre_Order = Iterator.Iterator_Kind
               then
               return;
            end if;
            --|
            --|3) it is the second time we arrive in this node as we are returning from the left
            --|   (current *2 = previous  )   go right
            --|
            elsif Iterator.Current_Index * 2 = Iterator.Previous_Index
            then
            Iterator.Previous_Index := Iterator.Current_Index;
            Iterator.Current_Index  := Iterator.Previous_Index * 2 + 1;

            if In_Order = Iterator.Iterator_Kind
               then
               return;
            end if;
            --|
            --|4) it is the third time we arrive in this node as we are returning from the right
            --|   (current *2 +1 = previous  )   go up
            --|
            elsif Iterator.Current_Index * 2 + 1 = Iterator.Previous_Index
            then
            Iterator.Previous_Index := Iterator.Current_Index;
            Iterator.Current_Index  := Iterator.Previous_Index / 2;

            if Post_Order = Iterator.Iterator_Kind
               then
               return;
            end if;
         end if;
      end loop;

   end To_Next_Value;
```

12.7.1.6.4 Fini

Noter que le test de fin post-order doit "tourner" un fois de plus pour visiter le sommet.

```
   function Is_Done(Iterator : in     Iterator_Type) return Boolean
      is
      begin

      if Post_Order = Iterator.Iterator_Kind
         then
         if Post_Order_Done
            then return True;
         end if;
         Post_Order_Done := Iterator.Current_Index = 0 and then Iterator.Previous_Index = 1;
         return False;

         else
         return Iterator.Current_Index = 0 and then Iterator.Previous_Index = 1;
      end if;
   end Is_Done;
```

12.7.2 L'arbre binaire: Implémentation avec des pointeurs

Beaucoup d'aspects de la partie programmation sont simplifiés, sauf l'itérateur qui doit être écrit actif (et cette fois c'est difficile !!!).

12.7.2.1 Données et Types de données:

La partie générique:

```
generic

type Data_Type is private;
--|
--| for = operator
--|
with function "=" (Left  : in Data_Type;
                   Right : in     Data_Type) return Boolean is <>;
```

```
  --|
  --| for > operator
  --|
  with function ">" (Left  : in Data_Type;
           Right : in    Data_Type) return Boolean is <>;

package Linked_Bin_Tree
    is

    Bin_Tree_Empty             : exception;
    Record_Not_Found_On_Delete : exception;

    type Iterator_Type is limited private;
    type Bin_Tree_Type is limited private;
  type Iterator_Kind_Type is (Pre_Order, In_Order, Post_Order);
.........
```

Limited private: pour **obliger l'utilisateur à utiliser les opérations du "package" et interdire l'assignation.**

```
    private

    type Node_Type;

    type Node_Access_Type is access all Node_Type;

    type Bin_Tree_Type is
       record
       Access_To_Top   : Node_Access_Type := null;
       How_Many_Stored : Integer          := 0;
    end record;

    type Node_Type is
       record
       Data  : Data_Type;
       Left  : Node_Access_Type := null;
       Right : Node_Access_Type := null;
    end record;
```

l'utilisation d'un stack est indispensable pour programmer une récursion et sera utilisé dans l'itérateur.

```
    package My_Stack is new Linked_Stack(Data_Type => Node_Access_Type);
    Iterator_Stack : My_Stack.Stack_Type;

    type Iterator_Type is
       record
       Current_Node_Access : Node_Access_Type   := null;
       Iterator_Kind       : Iterator_Kind_Type := In_Order;
       First_Time          : Boolean            := True;
       How_Many_Stored     : Natural            := 0;
       Current_Count       : Natural            := 0;
       On_Off              : Boolean            := True;
       Loop_Done           : Boolean            := False;
       Q                   : Node_Access_Type   := null;
    end record;

    procedure Free_Node is new Ada.Unchecked_Deallocation(Object => Node_Type,
                                                         Name   => Node_Access_Type);
```

Example d'instantiation:

```
    --|
    --|  data as a pointer to a string
    --|
    type My_Data_Type is access all String;

    function "=" (Left  : in My_Data_Type;
                  Right : in     My_Data_Type)
       return Boolean
       is
       begin
       return Left.all = Right.all;
    end "=";

    function ">" (Left  : in My_Data_Type;
                  Right : in     My_Data_Type)
       return Boolean
       is
       begin
       return Left.all > Right.all;
    end ">";

    package My_Bin_Tree is new Linked_Bin_Tree(Data_Type => My_Data_Type,
                                               "="       => "=",
                                               ">"       => ">");

    A_Bin_Tree              : My_Bin_Tree.Bin_Tree_Type;
```

```
   My_Iterator             : My_Bin_Tree.Iterator_Type;
```

Example d'utilisation:

```
   My_Bin_Tree.Clear(The_Bin_Tree => A_Bin_Tree);

   My_Bin_Tree.Insert(The_Data        => new String'("A"),
                      In_The_Bin_Tree => A_Bin_Tree);


   My_Bin_Tree.Seek(Data            => new String'("ABCDE"),
                    In_The_Bin_Tree => A_Bin_Tree,
                    Success         => Ok,
                    Found_Record    => My_Data);
```

Premier itérateur: pre-order

```
   My_Bin_Tree.Initialize(Iterator         => My_Iterator,
                          For_The_Bin_Tree => A_Bin_Tree,
                          Iterator_Kind    => My_Bin_Tree.Pre_Order);

   while not My_Bin_Tree.Is_Done(Iterator => My_Iterator) loop

      My_Data := My_Bin_Tree.Current_Value(Iterator => My_Iterator);
      Ada.Text_Io.Put(My_Data.all & ' ');

      My_Bin_Tree.To_Next_Value(Iterator => My_Iterator);
   end loop;
```

Second itérateur: in-order

```
   My_Bin_Tree.Initialize(Iterator         => My_Iterator,
                          For_The_Bin_Tree => A_Bin_Tree,
                          Iterator_Kind    => My_Bin_Tree.In_Order);

   while not My_Bin_Tree.Is_Done(Iterator => My_Iterator) loop

      My_Data := My_Bin_Tree.Current_Value(Iterator => My_Iterator);
      Ada.Text_Io.Put(My_Data.all & ' ');

      My_Bin_Tree.To_Next_Value(Iterator => My_Iterator);
   end loop;
```

Troisième itérateur: post-order

```
   My_Bin_Tree.Initialize(Iterator         => My_Iterator,
                          For_The_Bin_Tree => A_Bin_Tree,
                          Iterator_Kind    => My_Bin_Tree.Post_Order);

   while not My_Bin_Tree.Is_Done(Iterator => My_Iterator) loop

      My_Data := My_Bin_Tree.Current_Value(Iterator => My_Iterator);
      Ada.Text_Io.Put(My_Data.all & ' ');

      My_Bin_Tree.To_Next_Value(Iterator => My_Iterator);
   end loop;


     My_Bin_Tree.Delete(The_Data        => new String'(String_32),
                        In_The_Bin_Tree => A_Bin_Tree);
```

12.7.2.2 Ajout

Deux pointeurs auxiliaires sont nécessaires. : previous et current. L'ajout du premier élément est différent de l'ajout d'un nième comme dans beaucoup de structures avec pointeurs. Il faut également conserver la direction du sous arbre (Droit ou gauche pour l'ajout du nouvel élément dans l'arbre.

previous←pointeur du sommet
current←pointeur du sommet
Si premier élément :créer et attacher au pointeur de la structure vers le sommet et laisser vide les pointeurs gauches et droit..
Si pointeur du sommet est vide: créer un nouveau noeud, y mettre les données du nouvel élément , laisser vide les pointeurs gauche et droit et l'attacher au pointeur du sommet.
En ada:

```
Access_To_Top := new Node_Type'(Left  => null,
                                Data  => The_Data,
                                Right => null);
```

Le but est de trouver un emplacement vide pour mettre le nouvel élément.

Boucle tant que le noeud/feuille n'est pas vide (non égal)
tant que current ≠ null boucle
Garder le précédent
previous← current

Si la nouvelle clé est < que la clé du noeud courant
current.data < clé
direction← gauche
index←index.gauche
Aller au fils gauche
sinon
Aller au fils droit
direction← droite
index←index.droit
fin si
fin boucle
Insérer le nouvel élément
tableau(index) ← élément
Si la direction est à gauche, créer un nouveau noeud, y mettre les données du nouvel élément et l'attacher au pointeur GAUCHE du parent . **Noter l'utilisation de "previous" qui pointe sur le parent, c'est NÉCESSAIRE CAR IL N'Y A AUCUN MOYEN D'ACCÉDER AU PARENT À PARTIR DE SON FILS (CURRENT) !!!!!**
Si direction = gauche

```
Previous.left := new Node_Type'(Left  => null,
                                Data  => The_Data,
                                Right => null);
```

Sinon, créer un nouveau noeud, y mettre les données du nouvel élément et l'attacher au pointeur DROIT du parent . En ada:

```
Previous.right := new Node_Type'(Left  => null,
                                 Data  => The_Data,
                                 Right => null);
```

Fin si

Code:

```
procedure Insert(The_Data        : in     Data_Type;
                 In_The_Bin_Tree : in out Bin_Tree_Type)
   is
   type Right_Left_Type is (R, L);
   Right_Left            : Right_Left_Type  := Right_Left_Type'First;
   Link_To_Previous_Node : Node_Access_Type := In_The_Bin_Tree.Access_To_Top;
   Link_To_Current_Node  : Node_Access_Type := In_The_Bin_Tree.Access_To_Top;
   begin
   --|
   --| first node ??
   --|
   if null = Link_To_Current_Node
      then
      In_The_Bin_Tree.Access_To_Top := new Node_Type'(Left  => null,
                                                      Data  => The_Data,
                                                      Right => null);
      else
      --|
      --| not the first to be added: seek position to insert
      --|
      while null /= Link_To_Current_Node loop
         --|
         --| keep previous    S O   T H A T   Y O U   C A N   A D D !!!!
         --|
         Link_To_Previous_Node := Link_To_Current_Node;
         --|
         --| left or right
         --|
         if Link_To_Current_Node.Data > The_Data
            then

            Right_Left          := L;
            Link_To_Current_Node := Link_To_Current_Node.Left;
            else

            Link_To_Current_Node := Link_To_Current_Node.Right;
            Right_Left          := R;
         end if;
      end loop;
      --|
      --| position found do the insert
      --|
      if Right_Left = L
         then
         Link_To_Previous_Node.Left := new Node_Type'(Left  => null,
                                                      Data  => The_Data,
                                                      Right => null);
         else
         Link_To_Previous_Node.Right := new Node_Type'(Left  => null,
                                                       Data  => The_Data,
                                                       Right => null);
      end if;
   end if;
   --|
   --| keep track
   --|
   In_The_Bin_Tree.How_Many_Stored := 1  In_The_Bin_Tree.How_Many_Stored;
end Insert;
```

12.7.2.3 Délétion

L'idée est la même que dans le cas de l'implémentation dans un tableau. Ici, l'emploi de pointeurs simplifient les détails. Après avoir trouvé dans l'arbre l'élément à supprimer appelé A et conservé son origine (fils droit ou gauche du parent), il y a trois cas à prendre en compte: pas de fils, un seul fils ou deux fils:

1. L'élément A à supprimer est stocké dans une feuille, elle n'a donc pas de fils et sa suppression est simple: le contenu est remplacé par un pointeur vide.
2. L'élément A à supprimer est stocké dans un noeud qui n'a qu'un seul fils: on le remplace A par le sous arbre dont le sommet est le fils.
3. L'élément A à supprimer est stocké dans un noeud qui a deux fils:

   La première étape consiste a trouver:
- Ou bien son successeur in-order (appelons le B) qui est le fils le plus à gauche du sous arbre droit de A
- Ou bien son prédécesseur in-order (appelons le B) qui est le fils le plus à droite du sous arbre gauche de A.

   Comme aucun de ces noeuds ne peut avoir plus d'un fils sinon ils ne peuvent pas être le prédécesseur ou le successeur "in-order", il peut être supprimé en utilisant le cas numéro 2.
5. Une fois trouvé, on remplace A par le sous arbre dont B est le sommet et on supprime cet élément avec deux cas : il n'a pas de fils (cas numéro 1) ou il a un fils cas numéro 2).

Le problème se décompose en deux parties :
1. Recherche de l'élément à supprimer.
2. Effacement du noeud avec les trois cas décrits ci-dessus

12.7.2.3.1 Recherche avec direction depuis le parent

```
Commencer au sommet en faisant pointer un pointeur  auxiliaire sur le sommet de l'arbre
courant←structure.pointeur sur le sommet
Conservation du pointeur  courant et précédent
précédent ← courant
Boucle tant que le noeud/feuille n'est pas vide (non égal)
tant que auxiliaire  ≠  null  boucle
Si la nouvelle clé est < que la clé du noeud courant
si courant.data < clé
Aller au fils gauche
précédent ← courant
courant← courant.gauche
direction ← gauche
fin si
Si la nouvelle clé est > que la clé du noeud courant
si tableau(index) > clé
Aller au fils droit
précédent ← courant
courant← courant.droit
Direction ← droite
sinon trouvé
appel délétion du noeud
Sortie ok
fin si
fin boucle
Non trouvé
Sortie ERREUR
```

12.7.2.3.2 Suppression: le choix entre les QUATRE cas_

```
Test aucun fils
si courant.droit  = vide
et  courant.droit = vide
Test deux fils
si courant.droit  ≠  vide
et  courant.droit  ≠  vide
Test un fils gauche
```

si courant.droit = vide
et courant.droit ≠ vide
Sinon un fils droit

Le cas où il n'y a aucun fils

Récupération de la place mémoire. En ada:
```
Free_Node(X => Parent_Of_The_Node_To_Be_Deleted.Left);
```

Le cas où il n'y a un fils gauche

Appel de Remplacement par le sous arbre pour le noeud de l'élément trouvé, ici on manoeuvre les pointeurs en utilisant l'origine depuis le parent.

Mettre un pointeur auxiliaire
auxiliaire ← courant.droit

Attacher au parent du bon côté!!
si direction = gauche
parent .gauche ←auxiliaire
ou bien
parent .droit ←auxiliaire
si direction = gauche

Le cas où il y a deux fils.

Trouver le successeur in-order qui est le fils le plus à gauche du sous arbre droit de indice de l'élément à supprimer
Aller au fils gauche
courant ←courant.gauche
un fils droit ??
si courant.droit ≠ null
OUI: trouver le fils le plus à droite qui n'a pas de fils droit
Tant que courant.droit ≠ null boucle
courant ←successeur
successeur ← courant.droit
fin boucle
Remplacer le noeud à supprimer par le noeud trouvé ci-dessus
successeur.gauche ← noeud à supprimer.gauche
successeur.droit ← noeud à supprimer.droit
En utilisant la direction connecter au parent du noeud à effacer
si direction=gauche parent du noeud à effacer. Gauche pointe vers le successeur
parent du noeud à effacer .gauche ← successeur
sinon
parent du noeud à effacer .droit ← successeur
fin si
NON: il n'y a pas de fils droit
sinon
successeur ← Current;
successeur.droit ← noeud à effacer.droit
En utilisant la direction, connecter au parent du noeud à effacer
si direction=gauche parent du noeud à effacer. Gauche pointe vers le successeur
parent du noeud à effacer .gauche ← successeur
sinon
parent du noeud à effacer .droit ← successeur
fin si
Effacer et récupérer l'espace mémoire, en ada:
```
Free_Node(X => Node_To_Be_Deleted);
```

Code:

```
procedure Delete(The_Data        : in     Data_Type;
                 In_The_Bin_Tree : in out Bin_Tree_Type)
       is
       --
       --There are several cases to consider:
       --
       --1) Deleting a leaf: Deleting a node with no children is easy,
       --   as we can simply remove it from the tree.
       --
       --2) Deleting a node with one child: Delete it and replace it with its child.
       --
       --3) Deleting a node with two children:
       --   Suppose the node to be deleted is called Z.
```

```
--   We replace the value of Z with either its in-order successor
--   (the left-most child of the right subtree)
--   or the in-order predecessor (the right-most child of the left subtree).
--   Once we find either the in-order successor or predecessor,
--   swap it with Z, and then delete it.
--   Since either of these nodes must have less than two children
--   (otherwise it cannot be the in-order successor or predecessor),
--   it can be deleted using the previous two cases.
--
--  In a good implementation, it is generally recommended
--  to avoid consistently using one of these nodes,
--  because this can unbalance the tree  (to be done).

type Right_Left_Type is (R, L);

procedure Delete_A_Node(Node_To_Be_Deleted  : in out Node_Access_Type;
                        Parent_Of_The_Node_To_Be_Deleted : in out Node_Access_Type;
                        Side_From_The_Parent_To_The_Node_To_Be_Deleted : in out Right_Left_Type)
   is
   Successor : Node_Access_Type;
   Current   : Node_Access_Type := Node_To_Be_Deleted;
   begin
   --|
   --|============================================
   --| The Node to be deleted has no children
   --|============================================
   --|
   if Current.Right = null
      and then Current.Left = null
      then
      --|
      --|Clear the Appropriate Side of Parent
      --|
      if Side_From_The_Parent_To_The_Node_To_Be_Deleted = L
         then
         Free_Node(X => Parent_Of_The_Node_To_Be_Deleted.Left);
         else
         Free_Node(X => Parent_Of_The_Node_To_Be_Deleted.Right);
      end if;
      --|
      --|============================================================
      --| The Node To Be Deleted Has A Right Child But No Left Child
      --|============================================================
      --|
      elsif Current.Right /= null
      and then Current.Left = null
      then
      Successor := Current.Right;
      --|
      --|Attach Node To Parent
      --|
      if Side_From_The_Parent_To_The_Node_To_Be_Deleted = L
         then
         Parent_Of_The_Node_To_Be_Deleted.Left := Successor;
         else
         Parent_Of_The_Node_To_Be_Deleted.Right := Successor;
      end if;

      Free_Node(X => Current);
      --|
      --|============================================================
      --| The Node To Be Deleted Has A Left Child But No Right Child
      --|============================================================
      --|
      elsif Current.Right = null
      and then Current.Left /= null
      then
      Successor := Current.Left;
      --|
      --|Attach Node To Parent
      --|
      if Side_From_The_Parent_To_The_Node_To_Be_Deleted = L
         then
         Parent_Of_The_Node_To_Be_Deleted.Left := Successor;
         else
         Parent_Of_The_Node_To_Be_Deleted.Right := Successor;
      end if;

      Free_Node(X => Current);
      --|
      --|============================================
      --| Node To Be Deleted Has Two Children
      --|============================================
      --|
      elsif Current.Right /= null
      and then Current.Left /= null
      then
      --|
      --| Set Current To The Left Child of The Node To Be Deleted
      --|
      Current := Current.Left;
      if Current.Right /= null
         then
         Successor := Current.Right;
         --|
         --| walk Down To The Right end of The Subtree of The Left Child
         --| of The Node To Be Deleted
```

```
            --|
            while Current.Right /= null loop
               Current   := Successor;
               Successor := Current.Right;
            end loop;
            --|
            --| replace The Node To Be Deleted with The Node Found
            --| in this case :
            --| Attach left Child of Node To Be Deleted
            --|
            Current.Right   := null;
            Successor.Left  := Node_To_Be_Deleted.Left;
            Successor.Right := Node_To_Be_Deleted.Right;
            --|
            --| Attach Node To Parent
            --|
            if Side_From_The_Parent_To_The_Node_To_Be_Deleted = L
               then
               Parent_Of_The_Node_To_Be_Deleted.Left := Successor;
               else
               Parent_Of_The_Node_To_Be_Deleted.Right := Successor;
            end if;
            --|
            --| There is No Right Subtree because the right pointer is null :
            --| Replace The Node To Be Deleted with Its Left Child
            --| in this case :
            --| Attach Right Child of Node To Be Deleted
            --|
            else
            Successor       := Current;
            Successor.Right := Node_To_Be_Deleted.Right;
            --|
            --| Attach Node To Parent
            --|
            if Side_From_The_Parent_To_The_Node_To_Be_Deleted = L
               then
               Parent_Of_The_Node_To_Be_Deleted.Left := Successor;
               else
               Parent_Of_The_Node_To_Be_Deleted.Right := Successor;
            end if;
            Free_Node(X => Node_To_Be_Deleted);
         end if;
      end if;
   end Delete_A_Node;


   Right_Left                 : Right_Left_Type  := Right_Left_Type'First;
   Parent_Node                : Node_Access_Type := In_The_Bin_Tree.Access_To_Top;
   Current_Node               : Node_Access_Type := In_The_Bin_Tree.Access_To_Top;
   Found                      : Boolean          := False;
   Side_From_Parent_To_Current : Right_Left_Type  := Right_Left_Type'First;

   begin
   --|
   --| find the node to be removed
   --|
   while not (null = Current_Node
              or else Found) loop

      if Current_Node.Data = The_Data
         then
         Found := True;

         elsif The_Data > Current_Node.Data
         then
         Side_From_Parent_To_Current := R;
         Parent_Node                 := Current_Node;
         Current_Node                := Current_Node.Right;

         elsif Current_Node.Data > The_Data
         then
         Side_From_Parent_To_Current := L;
         Parent_Node                 := Current_Node;
         Current_Node                := Current_Node.Left;
      end if;
   end loop;
   --|
   --| ok remove else raise error
   --|
   if Found
      then
      Delete_A_Node(Node_To_Be_Deleted                           => Current_Node,
                    Parent_Of_The_Node_To_Be_Deleted             => Parent_Node,
                    Side_From_The_Parent_To_The_Node_To_Be_Deleted => Side_From_Parent_To_Current);
      else
      raise Record_Not_Found_On_Delete;
   end if;
   --|
   --| keep track
   --|
   In_The_Bin_Tree.How_Many_Stored := In_The_Bin_Tree.How_Many_Stored - 1;

end Delete;
```

12.7.2.4 Initialiser

Cette procédure parcours RÉCURSIVEMENT l'arbre en mode post-order jusqu'à visiter une feuille (POST ORDER => c'est le dernier accès à cette feuille), efface cette feuille et poursuit le parcours post-order classique avec sa double récursion.

procedure récursion(courant : pointeur)
si le pointeur courant est null retour récursion
si courant =null
retour
fin si
appel avec fils gauche
récursion(courant .gauche)
appel avec fils droit
récursion(courant .droit)
effacer le noeud courant c'est la dernière fois qu'il est visité (post-order)
En ada:
```
Free_Node(X => courant);
```
fin procedure
Premier appel pour démarrer la récursion avec courant pointant vers le sommet.
récursion(pointeur vers le sommet)

Code:

```
procedure Clear(The_Bin_Tree : in out Bin_Tree_Type)
       is
       --
       -- The procedure "Destroy_Tree"
       -- uses a    P O S T   O R D E R   w a l k    :it deletes every node it visits
       --           ============================
       --
       procedure Destroy_Tree(Link_To_Current_Node : in out Node_Access_Type)
          is
          begin
          --|
          --| end of branch  ?    go back up
          --|
          if null = Link_To_Current_Node
             then
             return;
          end if;
          --|
          --|  none of the above : keep walking the tree (customary POST ORDER walk)
          --|
          Destroy_Tree(Link_To_Current_Node => Link_To_Current_Node.Left);
          Destroy_Tree(Link_To_Current_Node => Link_To_Current_Node.Right);
          --|
          --| clear leaf
          --|
          Free_Node(X => Link_To_Current_Node);
       end Destroy_Tree;

       begin
       --|
       --| clear tree
       --|
       Destroy_Tree(Link_To_Current_Node => The_Bin_Tree.Access_To_Top);
       --|
       --| clear number
       --|
       The_Bin_Tree.How_Many_Stored := 0;
    end Clear;
```

12.7.2.5 Recherche

Commencer au sommet en faisant pointer un pointeur auxiliaire sur le sommet de l'arbre
auxiliaire←structure.pointeur sur le sommet
Boucle tant que le noeud/feuille n'est pas vide (non égal)
tant que auxiliaire ≠ null boucle
Si la nouvelle clé est < que la clé du noeud courant
si tableau(index) < clé
auxiliaire← auxiliaire.gauche
fin si
Aller au fils gauche
si tableau(index) > clé
Aller au fils droit
auxiliaire← auxiliaire.droit
sinon trouvé
fin si

fin boucle
Non trouvé

```
 procedure Seek(In_The_Bin_Tree : in     Bin_Tree_Type;
                   Data           : in     Data_Type;
                   Success        :    out Boolean;
                   Found_Record   :    out Data_Type)
       is
       Link_To_Current_Node : Node_Access_Type := In_The_Bin_Tree.Access_To_Top;
       begin
       --|
       --| kepp looking
       --|
       while null /= Link_To_Current_Node loop

          if Link_To_Current_Node.Data = Data
             then
             Found_Record := Link_To_Current_Node.Data;
             Success      := True;
             return;

             elsif Link_To_Current_Node.Data > Data
             then
             Link_To_Current_Node := Link_To_Current_Node.Left;

             else
             Link_To_Current_Node := Link_To_Current_Node.Right;
          end if;
       end loop;
       --|
       --| not found
       --|
       Success := False;
    end Seek;
```

12.7.2.6 Itérateur (pre-order, in-order & post-order)

12.7.2.6.1 Initialiser

```
procedure Initialize(Iterator         :    out Iterator_Type;
                        For_The_Bin_Tree : in     Bin_Tree_Type;
                        Iterator_Kind    : in     Iterator_Kind_Type := In_Order)
       is
       begin
       --|
       --| reset and initialize
       --|
       Iterator := (Current_Node_Access => For_The_Bin_Tree.Access_To_Top,
                    Iterator_Kind       => Iterator_Kind,
                    First_Time          => True,
                    How_Many_Stored     => For_The_Bin_Tree.How_Many_Stored,
                    Current_Count       => 0,
                    On_Off              => True,
                    Loop_Done           => False,
                    Q                   => null);
       --|
       --| clear stack
       --|
       My_Stack.Initialize(The_Stack => Iterator_Stack);
       --|
       --| get first data
       --|
       To_Next_Value(Iterator => Iterator);

    end Initialize;
```

12.7.2.6.2 Valeur

```
    function Current_Value(Iterator : in     Iterator_Type)
       return Data_Type
       is
       begin
       return Iterator.Current_Node_Access.Data;
    end Current_Value;
```

12.7.2.6.3 Suivant

Tous les détails sont dans le livre célèbre D. Knuth The Art of Computer Programming, Volume 1 page 320 Addison Wesley, le code lui même est extrêmement détaillé et commenté.

```
procedure To_Next_Value(Iterator : in out Iterator_Type)
   is
   begin
   --|
   --| three iterators  select the one needed
   --|
   case Iterator.Iterator_Kind
      is
      --|
      --|=========================================
      --| P R E   O R D E R   I T E R A T O R
      --|=========================================
      --|
      when Pre_Order =>
      --
      -- D. Knuth the art of computer programming Volume 1 page 320 Addison Wesley
      --
      -- T1: set stack empty
      --     p <- top of tree
      -- T2: if p=null go to T4
      -- T3: visit pre order
      --     push p in stack
      --     p <- p.left
      --     go to T2
      -- T4: if stack empty algo is over
      --     else p<- top of stack
      --     pop stack
      -- T5: p <- P.Right
      --     go to T2
      --
      In_Pre_Order_Walk : declare
      begin
         --|
         --| loop to go back up if needed
         --|
         Main : loop
            --|
            --| not to be done the first time: we go to the deepest left immediately
            --|
            if Iterator.First_Time
               then
               Iterator.On_Off        := False;
               Iterator.Current_Count := 1  Iterator.Current_Count;
               Iterator.First_Time    := False;
               exit Main;
            end if;
            --|
            --| the jump inside a loop is replaced by a test that's true 1/2 of the time
            --|
            while null /= Iterator.Current_Node_Access loop
               --|
               --| this time return and set flag
               --|
               if Iterator.On_Off
                  then
                  Iterator.Current_Count := 1  Iterator.Current_Count;
                  Iterator.On_Off        := False;
                  exit Main;
               end if;
               --|
               --|set flag and push pointer
               --|
               Iterator.On_Off := True;
               My_Stack.Push(Data         => Iterator.Current_Node_Access,
                             In_The_Stack => Iterator_Stack);

               Iterator.Current_Node_Access := Iterator.Current_Node_Access.Left;
            end loop;
            --|
            --|retrieve pointer
            --|
            Iterator.Current_Node_Access := My_Stack.Top_Of(The_Stack => Iterator_Stack);
            --|
            --|remove last used and go right
            --|
            My_Stack.Pop(The_Stack => Iterator_Stack);
            Iterator.Current_Node_Access := Iterator.Current_Node_Access.Right;
         end loop Main;
         --|
         --|catch popping an empty stack (it also could be done by a test)
         --|
         exception
         when Constraint_Error =>
         Iterator.Current_Count := 1  Iterator.Current_Count;
      end In_Pre_Order_Walk;
      --|
      --|=========================================
      --| I N    O R D E R   I T E R A T O R
      --|=========================================
      --|
      when In_Order =>
```

```
--
-- D. Knuth the art of computer programming Volume 1 page 320 Addison Wesley
--
-- T1: set stack empty
--     p <- top of tree
-- T2: if p=null go to T4
-- T3: push p in stack
--     p <- p.left
--     go to T2
-- T4: if stack empty algo is over
--     else p<- top of stack
--     pop stack
-- T5: visit in order
--     p <- P.Right
--     go to T2
--
In_Order_Walk : declare
begin
   --|
   --| not to be done the first time: we go to the deepest left immediately
   --|
   if not Iterator.First_Time
      then
      Iterator.Current_Node_Access := Iterator.Current_Node_Access.Right;
   end if;
   --|
   --| go as far to the left as you can, pushing pointers in the stack
   --|
   while not (null = Iterator.Current_Node_Access) loop

      My_Stack.Push(Data         => Iterator.Current_Node_Access,
                    In_The_Stack => Iterator_Stack);

      Iterator.Current_Node_Access := Iterator.Current_Node_Access.Left;
   end loop;
   --|
   --| we reached an empty pointer get back one step
   --|
   Iterator.Current_Node_Access := My_Stack.Top_Of(The_Stack => Iterator_Stack);
   --|
   --| remove the last visited one
   --|
   My_Stack.Pop(The_Stack => Iterator_Stack);
   --|
   --|set flag so that we go to the right at the next call to To_Next_Value
   --|
   Iterator.First_Time    := False;
   Iterator.Current_Count := 1  Iterator.Current_Count;
   --|
   --|catch popping an empty stack (it also could be done by a test)
   --|
   exception
   when Constraint_Error =>
   Iterator.Current_Count := 1  Iterator.Current_Count;
end In_Order_Walk;
--|
--|============================================
--| P O S T   O R D E R   I T E R A T O R
--|============================================
--|
when Post_Order =>
--
-- D. Knuth the art of computer programming Volume 1 page 565 Addison Wesley
--
-- T1: set stack empty
--     p <- top of tree
--     Q <- null
-- T2: if p=null go to T4
-- T3: push p in stack
--     p <- p.left
--     go to T2
-- T4: if stack empty algo is over
--     else p<- top of stack
--     pop stack
-- T5: if p.right = null or P.left =Q  got to T6
--     p <- P.Right
--     go to T2
-- T6: visit post_order
--     Q <- P
--     go to T4
--
Post_Order_Walk : declare
begin
   loop
      --|
      --| go as far to the left as you can, pushing pointers in the stack
      --|
      if Iterator.On_Off
         then
         while not (null = Iterator.Current_Node_Access) loop

            My_Stack.Push(Data         => Iterator.Current_Node_Access,
                          In_The_Stack => Iterator_Stack);

            Iterator.Current_Node_Access := Iterator.Current_Node_Access.Left;
         end loop;
      end if;
```

```
            --|
            --| we reached an empty pointer get back one step
            --|
            Iterator.Current_Node_Access := My_Stack.Top_Of(The_Stack => Iterator_Stack);
            --|
            --| back from right or right is null ????
            --|
            if (Iterator.Current_Node_Access.Right = null
                or else Iterator.Current_Node_Access.Right = Iterator.Q)
               then
               --|
               --| YES  keep Q for next time pop latest in stack and visit post_order
               --|
               Iterator.Q             := Iterator.Current_Node_Access;
               Iterator.Current_Count := 1  Iterator.Current_Count;
               Iterator.On_Off        := False;
               My_Stack.Pop(The_Stack => Iterator_Stack);
               exit;
               else
               --|
               --|NO use the loop to move in the tree
               --|
               Iterator.Current_Node_Access := Iterator.Current_Node_Access.Right;
               Iterator.On_Off              := True;
            end if;
         end loop;
         --|
         --|catch popping an empty stack (it also could be done by a test)
         --|
         exception
         when Constraint_Error =>
         Iterator.Current_Count := 1  Iterator.Current_Count;
      end Post_Order_Walk;
   end case;
end To_Next_Value;
```

12.7.2.6.4 Fini

```
function Is_Done(Iterator : in     Iterator_Type)
      return Boolean
      is
      begin
      return Iterator.Current_Count = 1  Iterator.How_Many_Stored;
   end Is_Done;
```

13 LES ALGORITHMES DE TRI CLASSIQUES

13.1 LE TRI BULLE (BUBBLE SORT)

Horreur !!!! , c'est un tri en O(N^2). Ne JAMAIS l'utiliser!!!!!
Néanmoins, pour des tableaux TRÈS PETITS ,....

Ce tri consiste à pousser la valeur la plus grande trouvée en parcourant le tableau à la dernière place. Soit le tableau T dimensionné de 1 à N:
Après le premier parcours (de 1 à N), la valeur la plus grande entre T(1) et T(N) est maintenant dans T(N).
Après le second parcours (de 1 à N-1), la valeur la plus grande entre T(1) et T(N-1) est maintenant dans T(N-1).
Après le parcours Numéro I (de 1 à I-1), la valeur la plus grande entre T(1) et T(I-1) est maintenant dans T(I-1)
...
soit N parcours à effectuer de N/2 en moyenne) le temps de calcul est donc proportionnel à N(N/2) soit ½ N^2. Il est donc proportionnel à N^2.

Pseudo code:

Boucle d'indice i de 1 à N
Boucle d'indice J de 1 à N-i
si T(J) > T(1+J) inverser T(J) et T(1+J)
fin boucle

fin boucle

13.2 LE TRI PAR TAS (HEAP SORT)

C'est un tri en O(N LOG N)

13.2.1 Algorithme

Le tri par tas est basé sur la relation tableau <=> arbre binaire et les propriétés des tas. Un tas est un arbre binaire ayant comme propriété la relation:
Valeur du parent > valeur des fils gauches et droits sans préciser de relation entre les fils gauches et droits.
Le stockage d'un arbre binaire dans un tableau se fait en utilisant la règle des indices qui est comme suit:

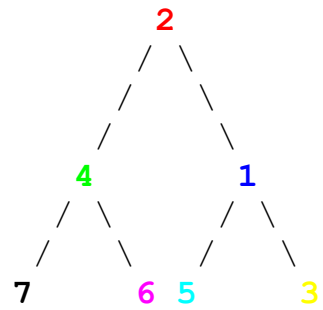
- indice fils gauche = 2* indice du parent
- indice fils droit = 2* indice du parent + 1
- indice du parent = indice du fils (gauche ou droit)/2

Cela permet une relation directe entre un arbre et un tableau. Le tri est basé sur le fait que le sommet est toujours l'élément dont la valeur est la plus élevée. Le tri commence par le tableau initial, qui en général, ne satisfait pas les propriétés d'un tas. On le transforme donc en un tas. Une fois cette transformation effectue le sommet (l'élément dont la valeur est la plus élevée) est changé pour le dernier élément du tableau. LA TAILLE DU TABLEAU EST DIMINUÉE D'UNE UNITE. L'arbre correspondant n'est donc plus un tas et est transformé à nouveau en un tas (seul le sommet doit être vérifié). La procédure est répétée et on obtient un tableau trié.

Illustration avec un tableau de 7 cases et l'arbre correspondant.

taille utilisée
7

indice	contenu
1	2
2	4
3	1
4	7
5	6
6	5
7	3



Cet arbre a une profondeur de 3 et contient 2^3-1 éléments. La transformation en tas commence au dernier élément à droite profondeur 2, c'est à dire la profondeur de l'arbre -1. L'indice correspondant est, par définition de l'équivalence tableaux <=> arbres binaires, 7/2=3 ou 7 est le nombre d'éléments contenus dans l'arbre. On appelle la procédure Move_In_Heap(parent=>3). C'est le début de l'algorithme « Move_In_Heap(parent) ».
On recherche le plus grand des trois éléments suivants: parent, fils gauche et fils droit:
il y a trois possibilités (si le fils correspondant n'existe pas on ne compare pas):

1. Le plus grand est le parent : Il n'y a rien a modifier
2. Le plus grand est le fils gauche : Parent et fils gauche sont échangés et le procédé recommence récursivement pour le fils gauche
3. Le plus grand est le fils droit : Parent et fils droit sont échangés et le procédé recommence récursivement pour le fils droit

PREMIERE PARTIE: TRANSFORMATION DE L'ARBRE DE DÉPART EN TAS

CONSTRUCTION DU TAS: ÉTAPE NUMERO 1

étape numéro 1.1: appel avec comme parent l'élément d'indice 3
Dans cet exemple, le parent d'indice 3 contient 1, le fils gauche d'indice 3 * 2 contient 5 et le fils droit d'indice 3 * 2 + 1 contient 3: c'est le fils gauche le plus grand.

étape numéro 1.1: échange du parent et du fils gauche:

```
|---------------|  ARBRE CORRESPONDANT AU TABLEAU en prenant en compte
|taille utilisée|  la taille utilisée
|       7       |
|---------------|                    2
|indice|contenu |                   / \
|  1   |   2    |                  /   \
|  2   |   4    |                 /     \
|  3   |   5    |                4       5
|  4   |   7    |               / \     / \
|  5   |   6    |              /   \   /   \
|  6   |   1    |             7     6 1     3
|  7   |   3    |
|---------------|
```

étape numéro 1.2: appel récursif avec comme parent le fils gauche,cet appel ne modifie rien car il n'y a aucun fils.

CONSTRUCTION DU TAS: ÉTAPE NUMERO 2

étape numéro 2.1: appel avec comme parent l'élément d'indice 2: le parent d'indice 2 contient 4, le fils gauche d'indice 2 * 2 contient 7 et le fis droit d'indice 2 * 2 + 1 contient 6: c'est le fils gauche le plus grand: échange du parent et du fils gauche.

étape numéro 2.1: échange du parent et du fils gauche

```
|---------------|  ARBRE CORRESPONDANT AU TABLEAU en prenant en compte
|taille utilisée|  la taille utilisée
|       7       |
|---------------|                    2
|indice|contenu |                   / \
|  1   |   2    |                  /   \
|  2   |   7    |                 /     \
|  3   |   5    |                7       5
|  4   |   4    |               / \     / \
|  5   |   6    |              /   \   /   \
|  6   |   1    |             4     6 1     3
|  7   |   3    |
|---------------|
```

étape numéro 2.2: appel récursif avec comme parent le fils gauche, cet appel ne modifie rien car il n'y a aucun fils.

CONSTRUCTION DU TAS: ÉTAPE NUMERO 3

étape numéro 3.1: appel avec comme parent l'élément d'indice 1, le parent d'indice 1 contient 2, le fils gauche d'indice 1 * 2 contient 7 et le fils droit d'indice 1 * 2 + 1 contient 5: c'est le fils gauche le plus grand: échange du parent et du fils gauche.

étape numéro 3.1: échange du parent et du fils gauche

```
|---------------|  ARBRE CORRESPONDANT AU TABLEAU en prenant en compte
|taille utilisée|  la taille utilisée
|       7       |
|---------------|                    7
|indice|contenu |                   / \
|  1   |   7    |                  /   \
|  2   |   2    |                 /     \
|  3   |   5    |                2       5
|  4   |   4    |               / \     / \
|  5   |   6    |              /   \   /   \
|  6   |   1    |             4     6 1     3
|  7   |   3    |
|---------------|
```

étape numéro 3.2: appel récursif avec comme parent le fils gauche, le parent d'indice 2 contient 2, le fils gauche d'indice 2 * 2 contient 4 et le fils droit d'indice 2 * 2 + 1 contient 6: c'est le fils droit le plus grand: échange du parent et du fils droit.

étape numéro 3.1: échange du parent et du fils droit.

```
|---------------|  ARBRE CORRESPONDANT AU TABLEAU en prenant en compte
|taille utilisée|  la taille utilisée
|       7       |
|---------------|                     7
|indice|contenu |                    / \
|  1   |   7    |                   /   \
|  2   |   6    |                  /     \
|  3   |   5    |                 6       5
|  4   |   4    |                / \     / \
|  5   |   2    |               /   \   /   \
|  6   |   1    |              4     2 1     3
|  7   |   3    |
|---------------|
```

étape numéro 3.3: appel récursif avec comme parent le fils droit: cet appel ne modifie rien car il n'y a aucun fils.

L'ARBRE EST MAINTENANT UN TAS

SECONDE PARTIE: ÉCHANGER LE CONTENU DE L'INDICE 1 ET DE LA TAILLE UTILISÉE et TRANSFORMER EN TAS

chaque étape comprend 3 opérations :
échange
diminution de la taille utilisée
transformation en tas (plus simple il n'y a qu'une modification)

ÉCHANGE ET DIMINUTION DE TAILLE: ÉTAPE NUMERO 1

étape numéro 1.1: échange et diminution de la taille

```
|---------------|  ARBRE CORRESPONDANT AU TABLEAU en prenant en compte
|taille utilisée|    L A   T A I L L E   U T I L I S E E
|       6       |
|---------------|                     3
|indice|contenu |                    / \
|  1   |   3    |                   /   \
|  2   |   6    |                  /     \
|  3   |   5    |                 6       5
|  4   |   4    |                / \     /
|  5   |   2    |               /   \   /
|  6   |   1    |              4     2 1
|  7   |   7    |
|---------------|
Le fils droit de 5 d'indice 7 n'est pas dessiné car la taille utilisée est de 6.
```

étape numéro 1.2: appel avec comme parent l'élément d'indice 1, le parent d'indice 1 contient 3, le fils gauche d'indice 1 * 2 contient 6 et le fis droit d'indice 1 * 2 + 1 contient 5: c'est le fils gauche le plus grand: échange du parent et du fils gauche.

```
|---------------|  ARBRE CORRESPONDANT AU TABLEAU en prenant en compte
|taille utilisée|    L A   T A I L L E   U T I L I S E E
|       6       |
|---------------|                     6
|indice|contenu |                    / \
|  1   |   6    |                   /   \
|  2   |   3    |                  /     \
|  3   |   5    |                 3       5
|  4   |   4    |                / \     /
|  5   |   2    |               /   \   /
|  6   |   1    |              4     2 1
|  7   |   7    |
|---------------|
```

étape numéro 1.3: appel récursif avec comme parent le fils gauche, le parent d'indice 2 contient 3 le fils gauche d'indice 2 * 2 contient 4 et le fis droit d'indice 2 * 2 + 1 contient 2: c'est le fils gauche le plus grand: change du parent et du fils gauche.

```
|---------------|  ARBRE CORRESPONDANT AU TABLEAU en prenant en compte
|taille utilisée|    L A   T A I L L E   U T I L I S E E
|       6       |
|---------------|                    6
|indice|contenu |                   / \
|  1   |   6    |                  /   \
|  2   |   4    |                 /     \
|  3   |   5    |                4       5
|  4   |   4    |               / \     /
|  5   |   3    |              /   \   /
|  6   |   1    |             3     2 1
|  7   |   7    |
|---------------|
```

étape numéro 1.3: appel récursif avec comme parent le fils gauche, cet appel ne modifie rien car il n'y a aucun fils.

ÉCHANGE ET DIMINUTION DE TAILLE: ÉTAPE NUMERO 2

étape numéro 2.1: change et diminution de la taille

```
|---------------|  ARBRE CORRESPONDANT AU TABLEAU en prenant en compte
|taille utilisée|    L A   T A I L L E   U T I L I S E E
|       5       |
|---------------|                    1
|indice|contenu |                   / \
|  1   |   1    |                  /   \
|  2   |   4    |                 /     \
|  3   |   5    |                4       5
|  4   |   4    |               / \
|  5   |   3    |              /   \
|  6   |   6    |             3     2
|  7   |   7    |
|---------------|
```

étape numéro 2.2: appel avec comme parent l'élément d'indice 1, le parent d'indice 1 contient 1, le fils gauche d'indice 1 * 2 contient 4 et le fis droit d'indice 1 * 2 + 1 contient 5: c'est le fils droit le plus grand: échange du parent et du fils droit.

```
|---------------|  ARBRE CORRESPONDANT AU TABLEAU en prenant en compte
|taille utilisée|    L A   T A I L L E   U T I L I S E E
|       5       |
|---------------|                    5
|indice|contenu |                   / \
|  1   |   5    |                  /   \
|  2   |   4    |                 /     \
|  3   |   1    |                4       1
|  4   |   4    |               / \
|  5   |   3    |              /   \
|  6   |   6    |             3     2
|  7   |   7    |
|---------------|
```

étape numéro 2.3: appel récursif avec comme parent le fils droit: cet appel ne modifie rien car il n'y a aucun fils.

ÉCHANGE ET DIMINUTION DE TAILLE: ÉTAPE NUMERO 3

étape numéro 3.1: échange et diminution de la taille

```
|---------------|  ARBRE CORRESPONDANT AU TABLEAU en prenant en compte
|taille utilisée|    L A   T A I L L E   U T I L I S E E
|       4       |
|---------------|                    2
|indice|contenu |                   / \
|  1   |   2    |                  /   \
|  2   |   4    |                 /     \
|  3   |   1    |                4       1
|  4   |   3    |               /
|  5   |   5    |              /
|  6   |   6    |             3
|  7   |   7    |
|---------------|
```

étape numéro 3.2: appel avec comme parent l'élément d'indice 1, le parent d'indice 1 contient 2, le fils gauche d'indice 1 * 2 contient 4 et le fis droit d'indice 1 * 2 + 1 contient 1: c'est le fils gauche le plus grand: échange du parent et du fils gauche

```
|---------------|  ARBRE CORRESPONDANT AU TABLEAU en prenant en compte
|taille utilisée|    L A   T A I L L E   U T I L I S E E
|       4       |
|---------------|                    4
|indice|contenu |                   / \
|  1   |   4    |                  /   \
|  2   |   2    |                 /     \
|  3   |   1    |                2       1
|  4   |   3    |               /
|  5   |   5    |              /
|  6   |   6    |             3
|  7   |   7    |
|---------------|
```

étape numéro 3.2: appel récursif avec comme parent le fils gauche, le parent d'indice 2 contient 2, le fils gauche d'indice 2 * 2 contient 3 et le fis droit d'indice 2 * 2 + 1 est vide: c'est le fils gauche le plus grand: change du parent et du fils gauche.

```
|---------------|  ARBRE CORRESPONDANT AU TABLEAU en prenant en compte
|taille utilisée|    L A   T A I L L E   U T I L I S E E
|       4       |
|---------------|                    4
|indice|contenu |                   / \
|  1   |   4    |                  /   \
|  2   |   3    |                 /     \
|  3   |   1    |                3       1
|  4   |   2    |               /
|  5   |   5    |              /
|  6   |   6    |             2
|  7   |   7    |
|---------------|
```

étape numéro 3.3: appel récursif avec comme parent le fils gauche: cet appel ne modifie rien car il n'y a aucun fils.

ÉCHANGE ET DIMINUTION DE TAILLE: ÉTAPE NUMERO 4

étape numéro 4.1: échange et diminution de la taille

```
|---------------|  ARBRE CORRESPONDANT AU TABLEAU en prenant en compte
|taille utilisée|    L A   T A I L L E   U T I L I S E E
|       3       |
|---------------|                    2
|indice|contenu |                   / \
|  1   |   2    |                  /   \
|  2   |   3    |                 /     \
|  3   |   1    |                3       1
|  4   |   4    |
|  5   |   5    |
|  6   |   6    |
|  7   |   7    |
|---------------|
```

étape numéro 4.2: appel avec comme parent l'élément d'indice 1, le parent d'indice 1 contient 2, le fils gauche d'indice 1 * 2 contient 3 et le fis droit d'indice 1 * 2 + 1 contient 1: c'est le fils gauche le plus grand: échange du parent et du fils gauche.

```
|---------------|  ARBRE CORRESPONDANT AU TABLEAU en prenant en compte
|taille utilisée|    L A   T A I L L E   U T I L I S E E
|       3       |
|---------------|                    3
|indice|contenu |                   / \
|  1   |   3    |                  /   \
|  2   |   2    |                 /     \
|  3   |   1    |                2       1
|  4   |   4    |
|  5   |   5    |
|  6   |   6    |
|  7   |   7    |
|---------------|
```

étape numéro 4.2: appel récursif avec comme parent le fils gauche: cet appel ne modifie rien car il n'y a aucun fils.

ÉCHANGE ET DIMINUTION DE TAILLE: ÉTAPE NUMERO 5

étape numéro 5.1: change et diminution de la taille

```
|---------------|  ARBRE CORRESPONDANT AU TABLEAU en prenant en compte
|taille utilisée|    L A   T A I L L E   U T I L I S E E
|       2       |
|---------------|                     1
|indice|contenu |                    /
|   1  |   1    |                   /
|   2  |   2    |                  /
|   3  |   3    |                 2
|   4  |   4    |
|   5  |   5    |
|   6  |   6    |
|   7  |   7    |
|---------------|
```

plus rien a modifier le tableau est trié

13.2.2 Pseudo code

13.2.2.1 Swap

C'est simplement un échange de deux éléments d'un tableau avec une variable auxiliaire. Son utilisation rends le programme plus lisible.

```
procedure Swap(A : in out A_Type;
               I : in     Index_Type;
               J : in     Index_Type)
   is
   Dummy : Data_Type;
   begin
   Dummy := A(I);
   A(I)  := A(J);
   A(J)  := Dummy;
end Swap;
```

13.2.2.2 transformation d'un sous arbre en tas

On recherche le plus grand des trois éléments suivants parent fils gauche et fils droit:
il y a trois possibilités (si le fils n'existe pas on ne compare pas):
le plus grand est le parent : il n'y a rien a modifier
le plus grand est le fils gauche : on change parent et fils gauche
on recommence récursivement pour le fils gauche
le plus grand est le fils droit : on change parent et fils droit
on recommence récursivement pour le fils droit.
Il se résume ainsi (A: tableau contenant l'arbre et I: indice du parent):

calcul indices fils gauche et droit

si le fils gauche existe et si il est plus grand que le parent
noter l'utilisation de « and then » qui évite de faire le second test si le premier est faux (ce qui correspond au cas ou il n'y a pas de fils.
index← index du fils gauche
fin si

si le fils droit existe et si il est plus grand que le parent
noter l'utilisation de « and then » qui évite de faire le second test si le premier est faux (ce qui correspond au cas ou il n'y a pas de fils.
index← index du fils droit
fin si

si l'index est différent de celui du parent
swap parent et fils
appel récursif sur l'indice du fils le plus grand
fin si

soit en résumé:

transformer_en_tas(A : tableau contenant l'arbre ,
I : indice du parent,
Taille: taille du tableau)

*Index_du_fils_gauche ←2*I*
*Index_du_fils_droit← 1+ 2*I*
Index_du_plus_grand ← I

si Index_du_fils_gauche <= Taille
et aussi A(Index_du_fils_gauche) > A(I)
Index_du_plus_grand ← Index_du_fils_gauche
fin si

si Index_du_fils_droit <= Taille
et aussi A(Index_du_fils_droit) > A(I)
Index_du_plus_grand ← Index_du_fils_droit
fin si

si I /= Index_du_plus_grand
échanger A(I) avec A(Index_du_plus_grand)
transformer_en_tas(A, Index_du_plus_grand, Taille)
fin si

Le code Ada :

```
procedure Move_In_Heap(A : in out A_Type;
                       I : in     Index_Type)
      is
      Index_Of_Left_Child   : constant Integer := 2 * I;
      Index_Of_Right_Child  : constant Integer := Index_Of_Left_Child + 1;
      Index_Of_Largest_Item : Index_Type       := I;
      begin

      if Index_Of_Left_Child <= Heap_Size
         and then A(Index_Of_Left_Child) > A(I)
         then
         Index_Of_Largest_Item := Index_Of_Left_Child;
      end if;

      if Index_Of_Right_Child <= Heap_Size
         and then A(Index_Of_Right_Child) > A(Index_Of_Largest_Item)
         then
         Index_Of_Largest_Item := Index_Of_Right_Child;
      end if;

      if Index_Of_Largest_Item /= I
         then

         Swap(A => A,
              I => I,
              J => Index_Of_Largest_Item);

         Move_In_Heap(A => A,
                      I => Index_Of_Largest_Item);
      end if;

   end Move_In_Heap;
```

13.2.2.3 Construction du tas à partir d'un arbre quelconque

C'est une simple boucle inversée de 1 à la taille de l'arbre / 2 qui appelle la procédure précédente.

```
   procedure Build_The_Heap(A : in out A_Type)
      is
      begin

      for I in reverse 1 .. A'Last / 2 loop

         Move_In_Heap(A => A,
                      I => I);
      end loop;

   end Build_The_Heap;
```

13.2.2.4 Tri à partir d'un tas

C'est une procédure simple : une boucle inversée de 2 à la taille du tableau. Il y a trois étapes:
swap du premier avec le dernier courant
diminution de 1 de la taille courantes
Appel pour ordonner le tas

```
procedure Sort(A : in out A_Type)
   is
   begin

   for I in reverse 2 .. A'Last loop

      Swap(A => A,
           I => 1,
           J => I);

      Heap_Size := Heap_Size - 1;

      Move_In_Heap(A => A,
                   I => 1);
   end loop;

end Sort;
```

13.2.2.5 Programme principal

On ne peut pas faire plus simple: ordonner le tas, trier le tas.

```
Build_The_Heap(A => A);
Sort(A           => A);
```

13.3 LE TRI FUSION (MELT SORT)

C'est un tri en N LOG N. L'algorithme est divisé en deux procédures, l'une d'entre elle est une récursion double, l'autre fusionne deux demi-tableaux déjà triés. Un stockage auxiliaire est nécessaire. Le rôle de la récursion est de découper le tableau initial en tableaux contigus et de les rassembler ensuite après le tri. Cet algorithme est basé sur:

- Un tableau contenant un élément est déjà trié.
- À partir de deux tableaux contigus triés on peut facilement générer un tableau trié.

Par exemple un tableau de 7 éléments.

13.3.1 Le Rôle de la double récursion

Le pseudo code de cette partie s'écrit:

Récursion (D,F)
Si D >= F sortie de la récursion
M ← (D + F) / 2
Récursion (D,M)
Récursion (1+M,F)
Appel tri fusion(D,F)

Au niveau de l'appel à tri fusion, les valeurs successives de D et F permettent le fonctionnement du tri. Par exemple, si on appelle Récursion(1,7), les appels successifs à tri fusion s'écrivent:
Tri Fusion(1, 2), Tri Fusion(3, 4), Tri Fusion(1, 4) , ... et Tri Fusion(1, 7)

13.3.2 La partie tri fusion

Soit le tableau initial suivant (et le tableau auxiliaire nécessaire de même taille)

Indice	1	2	3	4	5	6	7
Tableau initial	18	17	15	11	13	12	19
Tableau auxiliaire							

Appel avec D=1 et F =2

Indice	1	2	3	4	5	6	7
Tableau initial	18	17	15	11	13	12	19
Tableau auxiliaire							

Copie des éléments d'indice 1 .. 2 dans le tableau auxiliaire: AUX(D..F) ←T(D..F)

Indice	1	2	3	4	5	6	7
Tableau initial	18	17	15	11	13	12	19
Tableau auxiliaire	18	17					

Trois indices auxiliaires I, J et K sont utilisés. Ils sont initialisés : i ←D, j←1+(D+F)/2 et K←D
Comparaison du contenu du tableau auxiliaire AUX(I) avec AUX(J): AUX(J) est le plus petit il est recopié dans le tableau initial T(K) ← AUX(J), J←1+J et K←1+K

Indice	1	2	3	4	5	6	7
Tableau initial	17	17	15	11	13	12	19
Tableau auxiliaire	18	17					

J = F il n'y a plus de comparaisons possibles, Le tableau initial est rempli avec les valeurs non utilisées pour i. Ici il n'en reste qu'une:

Indice	1	2	3	4	5	6	7
Tableau initial	17	18	15	11	13	12	19
Tableau auxiliaire	18	17					

Appel avec D=3 et F =4

Indice	1	2	3	4	5	6	7
Tableau initial	17	18	15	11	13	12	19
Tableau auxiliaire	18	17					

Copie des éléments d'indice 3 .. 4 dans le tableau auxiliaire: AUX(D..F) ←T(D..F)

Indice	1	2	3	4	5	6	7
Tableau initial	17	18	15	11	13	12	19
Tableau auxiliaire	18	17	15	11			

Trois indices auxiliaires I, J et K sont utilisés. Ils sont initialisés : i ←D, j←1+(D+F)/2 et K←D

Comparaison du contenu du tableau auxiliaire AUX(I) avec AUX(J): AUX(J) est le plus petit il est recopié dans le tableau initial : T(K) ← AUX(J), J←1+J et K←1+K

Indice	1	2	3	4	5	6	7
Tableau initial	17	18	11	11	13	12	19
Tableau auxiliaire	18	17	15	11			

J = F il n'y a plus de comparaisons possibles, Le tableau initial est rempli avec les valeurs non utilisées pour i. Ici il n'en reste qu'une:

Indice	1	2	3	4	5	6	7
Tableau initial	17	18	11	15	13	12	19
Tableau auxiliaire	18	17	15	11			

Appel avec D=1 et F =4

Indice	1	2	3	4	5	6	7
Tableau initial	17	18	11	15	13	12	19
Tableau auxiliaire	18	17	15	11			

Copie des éléments d'indice 1 .. 4 dans le tableau auxiliaire: AUX(D..F) ←T(D..F)

Indice	1	2	3	4	5	6	7
Tableau initial	17	18	11	15	13	12	19
Tableau auxiliaire	17	18	11	15			

Trois indices auxiliaires I, J et K sont utilisés. Ils sont initialisés : i ←D, j←1+(D+F)/2 et K←D
Comparaison du contenu du tableau auxiliaire AUX(I) avec AUX(J): AUX(J) est le plus petit il est recopié dans le tableau initial: T(K) ← AUX(J), J←1+J et K←1+K

Indice	1	2	3	4	5	6	7
Tableau initial	11	18	11	15	13	12	19
Tableau auxiliaire	17	18	11	15			

Comparaison du contenu du tableau auxiliaire AUX(I) avec AUX(J): AUX(J) est le plus petit il est recopié dans le tableau initial: : T(K) ← AUX(J), J←1+J et K←1+K

Indice	1	2	3	4	5	6	7
Tableau initial	11	18	11	15	13	12	19
Tableau auxiliaire	17	18	11	15			

Comparaison du contenu du tableau auxiliaire AUX(I) avec AUX(J): AUX(J) est le plus petit il est recopié dans le tableau initial: : T(K) ← AUX(J), J←1+J et K←1+K

Indice	1	2	3	4	5	6	7
Tableau initial	11	15	11	15	13	12	19
Tableau auxiliaire	17	18	11	15			

J = F il n'y a plus de comparaisons possibles, Le tableau initial est rempli avec les valeurs non utilisées pour i. Ici il n'en reste deux:

Indice	1	2	3	4	5	6	7
Tableau initial	11	15	17	18	13	12	19
Tableau auxiliaire	17	18	11	15			

Et ainsi de suite jusqu'au dernier appel:

Appel avec D=1 et F =4

Indice	1	2	3	4	5	6	7
Tableau initial	11	15	17	18	12	13	19
Tableau auxiliaire	17	18	11	15	13	12	19

Copie des éléments d'indice 1 .. 7 dans le tableau auxiliaire: AUX(D..F) ←T(D..F)

Indice	1	2	3	4	5	6	7
Tableau initial	11	15	17	18	12	13	19
Tableau auxiliaire	11	15	17	18	12	13	19

Trois indices auxiliaires I, J et K sont utilisés. Ils sont initialisés : i ←D, j←1+(D+F)/2 et K←D
I=1, J=5 et K=1
Comparaison du contenu du tableau auxiliaire AUX(I) avec AUX(J): AUX(I) est le plus petit il est recopié dans le tableau initial: T(K) ← AUX(I), I←1+I et K←1+K

Indice	1	2	3	4	5	6	7
Tableau initial	11	15	17	18	12	13	19
Tableau auxiliaire	11	15	17	18	12	13	19

I=2, J=5 et K=2
Comparaison du contenu du tableau auxiliaire AUX(I) avec AUX(J): AUX(J) est le plus petit il est recopié dans le tableau initial: T(K) ← AUX(J), J←1+J et K←1+K

Indice	1	2	3	4	5	6	7
Tableau initial	11	12	17	18	12	13	19
Tableau auxiliaire	11	15	17	18	12	13	19

I=2, J=6 et K=3
Comparaison du contenu du tableau auxiliaire AUX(I) avec AUX(J): AUX(J) est le plus petit il est recopié dans le tableau initial: T(K) ← AUX(J), J←1+J et K←1+K

Indice	1	2	3	4	5	6	7
Tableau initial	11	12	13	18	12	13	19
Tableau auxiliaire	11	15	17	18	12	13	19

I=2, J=7 et K=4
Comparaison du contenu du tableau auxiliaire AUX(I) avec AUX(J): AUX(I) est le plus petit il est recopié dans le tableau initial: T(K) ← AUX(I), I←1+I et K←1+K

Indice	1	2	3	4	5	6	7
Tableau initial	11	12	13	15	12	13	19
Tableau auxiliaire	11	15	17	18	12	13	19

I=3, J=7 et K=5
Comparaison du contenu du tableau auxiliaire AUX(I) avec AUX(J): AUX(I) est le plus petit il est recopié dans le tableau initial: T(K) ← AUX(I), I←1+I et K←1+K

Indice	1	2	3	4	5	6	7
Tableau initial	11	12	13	15	17	13	19
Tableau auxiliaire	11	15	17	18	12	13	19

I=4, J=7 et K=6
Comparaison du contenu du tableau auxiliaire AUX(I) avec AUX(J): AUX(I) est le plus petit il est recopié dans le tableau initial: T(K) ← AUX(I), I←1+I et K←1+K

Indice	1	2	3	4	5	6	7
Tableau initial	11	12	13	15	17	18	19
Tableau auxiliaire	11	15	17	18	12	13	19

I=5 la boucle s'arrête, Toutes les valeurs de I ont été utilisées, le tri fusion est terminé pour ce niveau de récursion, de plus la récursion elle même est terminée et le tri complet est effectué. Le pseudo-code s'écrit:

```
Tri Fusion (D,F)

--INITIALISATION

I ← D
M← (D+F)/2
J ← 1+M
K← D
AUX(D .. F) ← T(D .. F)

-- BOUCLE SUR I ET J

tant que I <= M et J <= F boucle
        si AUX(I) <= AUX(J)
          alors
          T(K) ← AUX(I)
          I     ← I + 1
         ou alors
          T(K) ← AUX(J)
          J     ← J + 1
        fin si
    K ← K +1
fin boucle

--Remplissage avec les valeurs restantes si il y en a
```

tant que I <= M boucle
* T(K) ← AUX(I)*
* K ← K + 1;*
* I ← I + 1;*
fin boucle

13.3.3 Le code en Ada

Code

```
with Ada.Text_Io;
--|
--|               |_______________|
--|               |               |
--|               |    Exo_6      |
--|               |_______________|
--|               |               |
--|
procedure Exo_06
   is
   --| data and index types
   subtype Data_Type is Integer  range 1 .. 100;
   subtype Index_Type is Integer range 1 .. 7;
   --| array type
   type The_Array_Type is array (Index_Type range <>) of Data_Type;-- needed in this program
   --|
   --|               |_______________|
   --|               |               |
   --|               | To_Screen     |
   --|               |_______________|
   --|               |               |
   --|
   procedure To_Screen(The_Array : in     The_Array_Type)
      is
      begin
      for I in The_Array'Range loop
         Ada.Text_Io.Put("|i="
                         & Integer'Image(I)
                         & " A="
                         & Integer'Image(The_Array(I)));
      end loop;
      Ada.Text_Io.New_Line;
   end To_Screen;
   --|
   --|               |_______________|
   --|               |               |
   --|               |     Merge     |
   --|               |_______________|
   --|               |               |
   --|
   procedure Merge(Array_To_Sort : in out The_Array_Type)
      is
      I               : Index_Type                                    := Array_To_Sort'First;
      Middle          : constant Index_Type                           := (Array_To_Sort'First + Array_To_Sort'Last) / 2;
      J               : Integer                                       := 1 + Middle;
      K               : Integer                                       := Array_To_Sort'First;
      Temporary_Array : constant The_Array_Type(Array_To_Sort'Range) := Array_To_Sort;
      begin
      Ada.Text_Io.Put_Line("Merge called with"
                           & Integer'Image(Array_To_Sort'First)
                           & " .."
                           & Integer'Image(Array_To_Sort'Last));
      --|
      --| merge the two sorted lists ( from both recursive calls ) :
      --|   Array_To_Sort(Low       .. middle)     and
      --|   Array_To_Sort(middle +1 .. High)  into one
      --|   Array_To_Sort(low       .. high)
      --| while loop because I is used after the end of the loop
      --|
      while I <= Middle and then J <= Array_To_Sort'Last loop
         --|
         --| store lowest one of Temporary_Array(i) or Temporary_Array(J)
         --| in Array_To_Sort(K)
         --| then increment index of the lowest side ( i or J)
         --| and increment k (storage index)
         --|
         if Temporary_Array(I) <= Temporary_Array(J)
            then
            Array_To_Sort(K) := Temporary_Array(I);
            I                := I + 1;
            else
            Array_To_Sort(K) := Temporary_Array(J);
            J                := J + 1;
         end if;
         --|
         --|  increment storage index
         --|
         K := K + 1;
      end loop;
      --|
```

```
      --|  fill in data not used if any
      --|
      while I <= Middle loop
         Array_To_Sort(K) := Temporary_Array(I);
         K                := K + 1;
         I                := I + 1;
      end loop;
   end Merge;
   --|
   --|               |_______________|
   --|               |               |
   --|               | Merge_Sort    |
   --|               |_______________|
   --|               |               |
   --|
   procedure Merge_Sort(Array_To_Sort : in out The_Array_Type)
      is
      Start_Index : constant Index_Type := Array_To_Sort'First;
      End_Index   : constant Index_Type := Array_To_Sort'Last;
      Middle      : constant Index_Type := (Start_Index + End_Index) / 2;
      begin
      Ada.Text_Io.Put_Line("Merge_Sort starting recurtion with Start_Index "
                           & Integer'Image(Start_Index)
                           & " End_Index"
                           & Integer'Image(End_Index));
      --|
      --|   recursion ends ??
      --|
      if Start_Index >= End_Index
         then
         Ada.Text_Io.Put_Line("Merge_Sort end recursion ");
         return;
      end if;
      --|
      --|  recursive call on lower half
      --|
      Merge_Sort(Array_To_Sort => Array_To_Sort(Start_Index .. Middle));
      --|
      --| recursive call on higher half
      --|
      Merge_Sort(Array_To_Sort => Array_To_Sort(Middle + 1 .. End_Index));
      --|
      --|   Merge the two sorted lists ( from both recursive calls ) :
      --|   Array_To_Sort(Low       .. middle)     and
      --|   Array_To_Sort(middle +1 .. High)  into one
      --|   Array_To_Sort(low       .. high)
      --|
      Merge(Array_To_Sort => Array_To_Sort);
   end Merge_Sort;
   --|*********************************************************************
   --|*******************      M A I N    *******************************
   --|*********************************************************************

   --|
   --| data entry (sort of)
   --|
   The_Array : The_Array_Type(Index_Type'Range) := (18, 14, 16, 13, 15, 12, 17);
   begin
   --|
   --|   call the sorting algo
   --|
   Merge_Sort(Array_To_Sort => The_Array);
   --|
   --|  result to screen
   --|
   To_Screen(The_Array => The_Array);
end Exo_06;
```

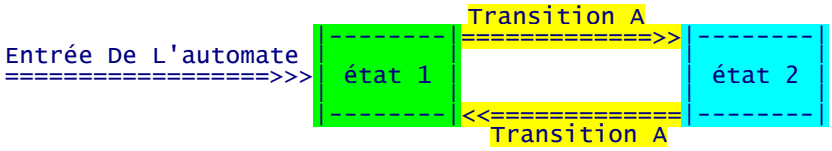
13.4 INTRODUCTION AUX GRAMMAIRES ET LANGAGE FORMEL, AUTOMATES À ÉTATS FINIS

La grammaire permet la définition d'un langage (pris au sens très large). Elle formalise toutes les relations syntaxiques et le vocabulaire.

13.4.1 Implémentation par un automate

Un automate est basé sur des notions complémentaires:
L'état
La transition
Toutes les transitions sortant d'un état sont différentes: automate déterministe (dit automate à états finis).

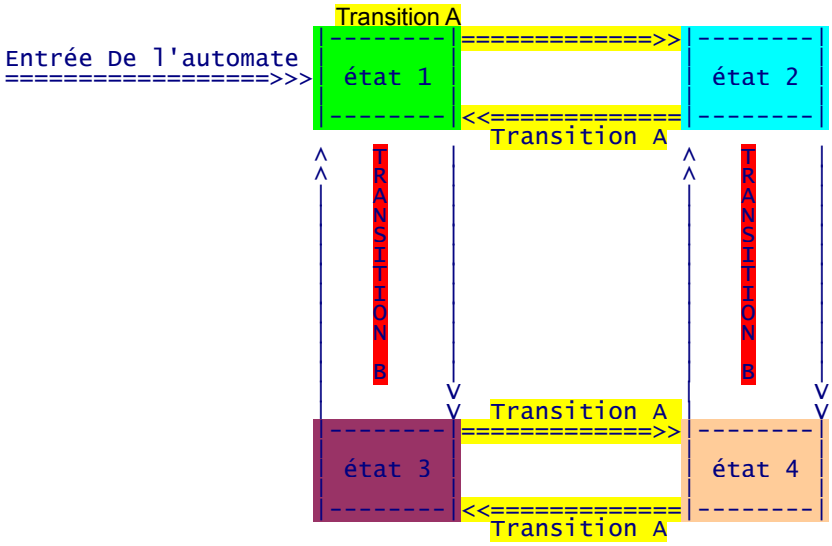
Prenons un exemple pour illustrer ces concepts: déterminer si un nombre d'objets est pair ou impair. Les objets sont placés sur deux tas (vides au début). Le premier objet est déposé sur le tas numéro 1 le second sur le tas numéro 2 , etc. Si le dernier objet est déposé sur le tas 1 le nombre d'objets est impair, sinon le nombre est pair. Appelons le fait de déposer l'objet sur un tas la transition A, le tas 1 l'état 1 et le tas 2 l'état 2.



Au début, l'automate est dans l'état 1
Le dépôt d'un objet, traduit en transition A modifie l'état de l'automate
L'automate est dans l'état 2
Le dépôt d'un objet, traduit en transition A modifie l'état de l'automate
L'automate est dans l'état 2
.....
Si l'état final de l'automate est l'état 1 le nombre d'objets est pair sinon le nombre d'objets est pair. Remarquez bien que le nombre total d'objets n'est pas connu. Ce n'était pas le problème posé.

Compliquons l'exemple avec deux types d'objets et la même question pair/impair: Il y a 4 possibilités!

Le dessin de l'automate se complique également. Il y a maintenant 4 états et 2 transitions (une pour chaque type d'objet).



Il y a maintenant 2 transitions différentes partant de chaque état:
transition A pour les objets de type A
transition B pour les objets de type B.

Au début, l'automate est dans l'état 1
Le dépôt d'un objet A, traduit en transition A modifie l'état de l'automate
L'automate est dans l'état 2
Le dépôt d'un objet B, traduit en transition B modifie l'état de l'automate
L'automate est dans l'état 4
Le dépôt d'un objet A, traduit en transition A modifie l'état de l'automate
L'automate est dans l'état 3
Le dépôt d'un objet B, traduit en transition B modifie l'état de l'automate
L'automate est dans l'état 1
.....

Quant tous les objets on été traités: l'automate s'arrête sur l'état final :
L'état 1: les objets A sont en nombre pair et les objets B sont en nombre pair
L'état 2: les objets A sont en nombre impair et les objets B sont en nombre pair
L'état 3: les objets A sont en nombre pair et les objets B sont en nombre impair
L'état 4: les objets A sont en nombre impair et les objets B sont en nombre impair

traduction d'un automate en code

Reprenons l'exemple précédent il faut:
le dessin de l'automate
une série d'objet a et b par exemple dans un texte les lettres aA et bB
une représentation pour les deux transitions
une représentation pour les 4 états
un représentation du dessin de l'automate
un algorithme

13.4.1.1 Le problème

Validation d'une expression arithmétique:

EXPRESSION ::= [ADDING_OPERATOR] TERM { ADDING_OPERATOR TERM}

TERM ::= FACTOR { MULTIPLYING_OPERATOR [ADDING_OPERATOR] FACTOR}

FACTOR ::= NUMBER | **'('** EXPRESSION **')'**

ADDING_OPERATOR ::= **'+'** | **'-'**

MULTIPLYING_OPERATOR ::= **'*'** | **'/'** | **'^'**

NUMBER ::= DIGIT { DIGIT } [**'.'** DIGIT { DIGIT }]

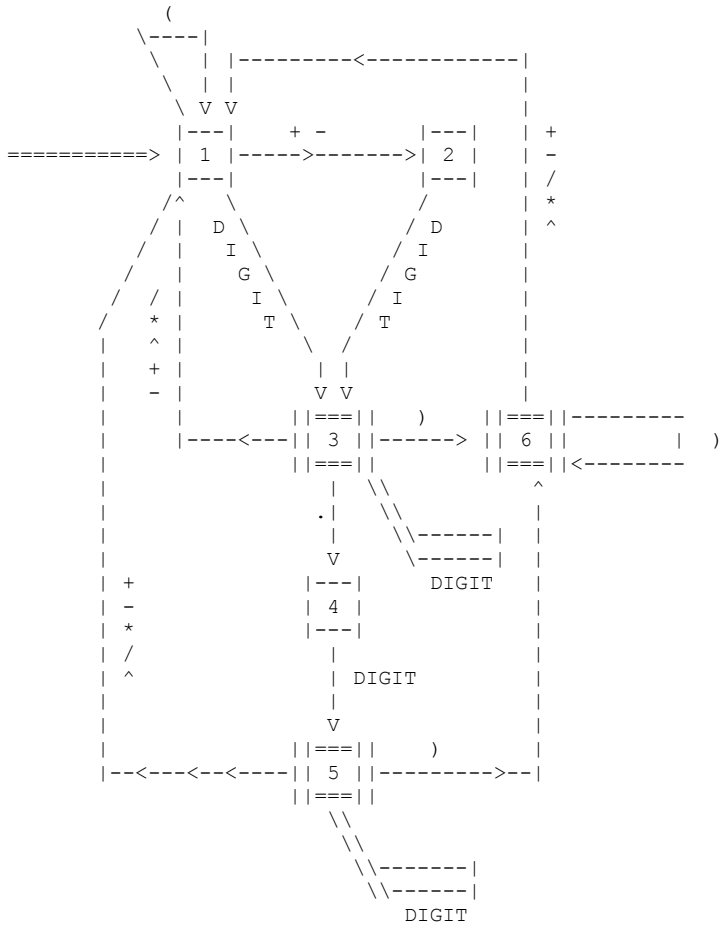
DIGIT ::= **'0'** | **'1'** | **'2'** | **'3'** | **'4'** | **'5'** | **'6'** | **'7'** | **'8'** | **'9'**

Exemples :

10 + 311 * 54 - (47 / 85)
1127 / 1283 + (45 * 87)
- 14 * 78 + (588 – 78) / 32^41.5
21 * 87 + 8 ^ -23.5

Informations :
Les abréviations (généralement utilisées en algorithmique pour décrire une grammaire) sont les suivantes:

{ xxxxxx }	veut dire 0 à n répétitions de xxxxxx
[yyyyyyy]	veut dire yyyyyy est optionnel
|	veut dire **ou**
::=	veut dire défini par
'xyz'	xyz sont des caractères obligatoires



3, 5 and 6 are final states

13.4.1.2 La solution

La traduction de l'automate ci-dessus :

```
with Ada.Text_Io;
procedure Test_Arithmetic
   is
```

Les transitions

```
   type Transition_Type is
   (Digit, Left_Parenthesis, Right_Parenthesis, Dot, Plus, Minus, Divide, Multiply, Power, Unknown);
```

Il y à 6 états avec l'état 0 représentant l'état erreur

```
   type State_Type is new Integer range 0 .. 6;
   type Matrix_Type is array (Transition_Type'Range, State_Type'Range) of State_Type;
   type Look_Up_Table_Type is array (Character'Range) of Transition_Type;
```

traduction de caractère à transition avec une table de vérité

```
   Look_Up_Table : constant Look_Up_Table_Type := ('0' .. '9'=> Digit,
                                                  '('      => Left_Parenthesis,
                                                  ')'      => Right_Parenthesis,
                                                  '.'      => Dot,
                                                  '+'      => Plus,
                                                  '-'      => Minus,
                                                  '/'      => Divide,
                                                  '*'      => Multiply,
                                                  '^'      => Power,
                                                  others   => Unknown);
```

la matrice représentant l'automate

```
   --                                                  0  1  2  3  4  5  6
   Matrix : constant Matrix_Type := (Digit            => (0, 3, 3, 3, 5, 5, 0),
                                     Left_Parenthesis => (0, 1, 0, 0, 0, 0, 0),
                                     Right_Parenthesis => (0, 0, 0, 6, 0, 6, 6),
                                     Dot              => (0, 0, 0, 4, 0, 0, 0),
                                     Plus             => (0, 2, 0, 1, 0, 1, 1),
                                     Minus            => (0, 2, 0, 1, 0, 1, 1),
                                     Divide           => (0, 0, 0, 1, 0, 1, 1),
                                     Multiply         => (0, 0, 0, 1, 0, 1, 1),
                                     Power            => (0, 0, 0, 1, 0, 1, 1),
                                     Unknown          => (0, 0, 0, 0, 0, 0, 1));
```

```
l'état initial est à 1 c'est l'entrée dans l'automate
    State             : State_Type      := 1;
    Transition        : Transition_Type := Transition_Type'First;
    Current_Character : Character       := ' ';
    User_S_Entry      : String(1 .. 50) := (others => ' ');
    Last              : Integer         := 0;

    begin
debug avec une entrée en dur dans le code
    Last                   := 31;
    User_S_Entry(1 .. Last) := "2+(3/54.8*(4-7)^3.543*-5*(5-4))";

    for I in 1 .. Last loop
lecture du caractère courant
       Current_Character := User_S_Entry(I);
traduction du caractère courant en transition
       Transition        := Look_Up_Table(Current_Character);
avance à l'état suivant en utilisant la matric qui représente l'automate
       State             := Matrix(Transition, State);
    end loop;
le résultat est ok pour l'état final 3,4 et 6
    Ada.Text_Io.Put(State_Type'Image(State));
  end Test_Arithmetic;
```

Le code complet est page: 281

13.4.2 Implémentation par une série de fonctions d'un exemple récursif

13.4.2.1 Le problème

C'est le même mais son implémentation est réalisée différemment:
l'entrée utilisateur est "découpée" en "tokens" qui sont sont les caractères de l'expression à tester.
L'accès à ces *token* se fait en utilisant les opérations: value, next, init et fini:
Chaque ligne de la grammaire est traduite par une fonction (généralement récursive) qui renvoie vrai faux selon que cette ligne est vraie ou fausse.

13.4.2.2 La solution

13.4.2.2.1 Les types de données

```
subtype User_Entry_Type is String(1 .. 100);

type Data_Type is
   record
   Chaine                      : User_Entry_Type := (others => ' ');
   Result                      : Boolean         := True;
   Position_Of_Current_Character : Integer       := 1;
   Position_Of_Last_Character    : Integer       := 0;
end record;

Data : Data_Type;
```

13.4.2.2.2 L'objet token et les opérations associées

```
lecture terminée

function Reading_Over
   return Boolean
   is
   begin

   return Data.Position_Of_Last_Character + 1 = Data.Position_Of_Current_Character;
end Reading_Over;
```

```
passe au token suivant

procedure Next
   is
```

```
      begin
      Data.Position_Of_Current_Character := 1 + Data.Position_Of_Current_Character;
   end Next;
```

lecture entrée utilisateur

```
   procedure Get_Data(Header : in     String;
                      Data   : in out Data_Type)
      is
      begin
      Ada.Text_Io.Put(Header & ' ');
      Ada.Text_Io.Get_Line(Data.Chaine, Data.Position_Of_Last_Character);
      exception
      when others =>
      Data.Chaine                     := (others => ' ');
      Data.Position_Of_Last_Character := 0;
   end Get_Data;
```

lecture du token courant

```
   function Current_Character
      return Character
      is
      begin
      return Data.Chaine(Data.Position_Of_Current_Character);
   end Current_Character;
```

Les fonctions sont déclarées avant pour éviter les problèmes d'ordre d'écriture

```
   function Expression return Boolean;
   function Term return Boolean;
   function Factor return Boolean;
   function Adding_Operator return Boolean;
   function Multiplying_Operator return Boolean;
   function Number return Boolean;
   function Digit return Boolean;
```

Les fonctions ont le nom utilisé dans la grammaire pour faciliter la lecture du programme:
EXPRESSION ::= [ADDING_OPERATOR] TERM { ADDING_OPERATOR TERM}

```
   function Expression return Boolean
      is
      Dummy : Boolean;
      begin
      Dummy := Adding_Operator;
      if not Term
         then
         return False;
      end if;
      while Adding_Operator loop
         if not Term
            then
            return False;
         end if;
      end loop;
      return True;
   end Expression;
```

TERM ::= FACTOR { MULTIPLYING_OPERATOR [ADDING_OPERATOR] FACTOR}

```
   function Term return Boolean
      is
      Dummy : Boolean;
      begin
      if not Factor
         then
         return False;
      end if;
      while Multiplying_Operator loop
         Dummy := Adding_Operator;
         if not Factor
            then
            return False;
         end if;
      end loop;
      return True;
   end Term;
```

FACTOR ::= NUMBER | '(' EXPRESSION ')'

```
   function Factor return Boolean
      is
      begin
      if Number
         then
         return True;
         elsif Current_Character = '('
         then
         Next;
         if not Expression
            then
            return False;
         end if;
         if Current_Character = ')'
            then
            Next;
            return True;
```

```
         end if;
      end if;
      return False;
   end Factor;
```

ADDING_OPERATOR ::= '+' | '-'

```
   function Adding_Operator return Boolean
      is
      begin
      if Current_Character = '+'
         or else Current_Character = '-'
         then
         Next;
         return True;
      end if;
      return False;
   end Adding_Operator;
```

MULTIPLYING_OPERATOR ::= '' | '/' | '^'*

```
   function Multiplying_Operator return Boolean
      is
      begin
      if Current_Character = '*'
         or else Current_Character = '/'
         or else Current_Character = '^'
         then
         Next;
         return True;
      end if;
      return False;
   end Multiplying_Operator;
```

NUMBER ::= DIGIT { DIGIT } ['.' DIGIT { DIGIT }]

```
   function Number return Boolean
      is
      begin
      if not Digit
         then
         return False;
      end if;
      while Digit loop
         null;
      end loop;
      if Current_Character = '.'
         then
         Next;
         if not Digit
            then
            return False;
         end if;
         while Digit loop
            null;
         end loop;
      end if;
      return True;
   end Number;
```

DIGIT ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

```
   function Digit return Boolean
      is
      begin
      if Current_Character in '0' .. '9'
      then
         Next;
         return True;
         else
         return False;
      end if;
   end Digit;
```

le résultat s'obtient très simlement:

```
   Data.Result := Data.Result and Reading_Over;
```

Le code complet est page:283

13.5 L'ARBRE DICTIONNAIRE

13.5.0.1 Données et Types de données:

L

Exemples d'utilisation:

le code est page 286

13.6 TIRAGE AU HASARD

13.6.1 Utilisation des packages inclus dans le langage

L'exemple suivant illustre l'utilisation des packages inclus dans le langage pour le tirage au hasard de différent type de données. La distribution est uniforme (chaque valeur à la même probabilité). Ce package est particulièrement utile pour tester des programme en générant un grand nombre de données.
Le fichier rnd_data.ads

```
package Rnd_Data is
   function Random    return String;
   function Random    return Integer;
   function Random    return Long_Integer;
   function Random    return Float;
   function Random    return Long_Float;
-- sert à initialiser la longueur des chaînes de caractéres
 procedure Init(Generated_String_Length : in     Positive := 10);
   private
   String_Length : Positive := 10;
end Rnd_Data;
```

Le fichier rnd_data.adb

```
with Ada.Numerics.Discrete_Random;
with Ada.Numerics.Float_Random;

package body Rnd_Data
   is
-- définition des longueurs et du type de caractére pour les chaînes
   subtype Word_Lenght_Type is Integer   range 1   .. 120;
   subtype Normal_Char_Type is Character range 'A' .. 'Z';
-- instantiation des divers module de tirage au sort
package Random_Word_Lenght is new Ada.Numerics.Discrete_Random(Result_Subtype => Word_Lenght_Type);

package Random_Character_Value is new Ada.Numerics.Discrete_Random(Result_Subtype => Normal_Char_Type);

package Random_Integer is new Ada.Numerics.Discrete_Random(Result_Subtype => Integer);

package Random_Long_Integer is new Ada.Numerics.Discrete_Random(Result_Subtype => Long_Integer);
-- ce module sert à initialiser à chaque fois les autres molules pour que le résultat ne soit pas toujours le même.
package Randomize_Seeds is new Ada.Numerics.Discrete_Random(Result_Subtype => Integer);
-- les générateurs
   G_Float                : Ada.Numerics.Float_Random.Generator;
   G_Char                 : Random_Character_Value.Generator;
   G_Word                 : Random_Word_Lenght.Generator;
   G_Integer              : Random_Integer.Generator;
   G_Long_Integer         : Random_Long_Integer.Generator;
   Random_Seeds_Generator : Randomize_Seeds.Generator;
   Dummy_String           : String(Word_Lenght_Type'Range);

   procedure Init(Generated_String_Length : in     Positive := 10)
      is
      begin
      Rnd_Data.String_Length := Generated_String_Length;
      --|
      --|    initialize seed randomization, this is   M A N D A T O R Y    to insure independent
```

```
      --|    seeding of all generators
      --|
      Randomize_Seeds.Reset(Gen => Random_Seeds_Generator);
      --|
      --| initialize all with a random seed
      --|
      Random_Character_Value.Reset(Gen       => G_Char,
                                   Initiator => Randomize_Seeds.Random
                                   (Gen      => Random_Seeds_Generator));

      Random_Word_Lenght.Reset(Gen       => G_Word,
                               Initiator => Randomize_Seeds.Random
                               (Gen      => Random_Seeds_Generator));

      Random_Integer.Reset(Gen       => G_Integer,
                           Initiator => Randomize_Seeds.Random
                           (Gen      => Random_Seeds_Generator));

      Random_Long_Integer.Reset(Gen       => G_Long_Integer,
                                Initiator => Randomize_Seeds.Random
                                (Gen      => Random_Seeds_Generator));

      Ada.Numerics.Float_Random.Reset(Gen       => G_Float,
                                      Initiator => Randomize_Seeds.Random
                                      (Gen      => Random_Seeds_Generator));
   end Init;

   function Random
      return String
      is
      begin
      --Word_Lenght  := Random_Word_Lenght.Random(Gen => G_Word);
      Dummy_String := (Dummy_String'Range => ' ');

      for K in 1 .. String_Length loop
         Dummy_String(K) := Random_Character_Value.Random(Gen => G_Char);
      end loop;

      return Dummy_String(1 .. String_Length);

   end Random;

   function Random
      return Integer
      is
      begin
      return Random_Integer.Random(Gen => G_Integer);
   end Random;

   function Random
      return Long_Integer
      is
      begin
      return Random_Long_Integer.Random(Gen => G_Long_Integer);
   end Random;

   function Random
      return Float
      is
      X : Float;
      K : Integer;

      begin

      K := Random mod 37;
      -- K := Integer'Max(K, - K);
      X := Ada.Numerics.Float_Random.Random(Gen => G_Float);
      X := X * 10.0 ** K;

      return X;
   end Random;

   function Random
      return Long_Float
      is
      X : Long_Float;
      K : Integer;

      begin

      K := Random mod 307;
      -- K := Integer'Max(K, - K);
      X := Long_Float(Ada.Numerics.Float_Random.Random(Gen => G_Float));
      X := X * 10.0 ** K;

      return X;
   end Random;

end Rnd_Data;
```

13.6.2 Utilisation d'un algorithme particulier: "mersenne twister"

L'algorithme utilisé garanti un trés bon tirage au sort Il est divisé en plusieurs parties Tirage au sort selon une distribution uniforme, puis transformation en un distribution normale.

Le fichier random_normal.ads

```
package Random_Normal
   is
-- tirage au sort selon une distribution normale de moyenne Mean et deviation standard Standard_deviation
   function Draw_Random(Mean               : in     Long_Float;
                        Standard_Deviation : in     Long_Float;
                        Max                : in     Long_Float := Long_Float'Last;
                        Min                : in     Long_Float := Long_Float'First)
   return Long_Float;

   private

   subtype Index_Type is Integer range 0 .. 623;
   type Mod_32_Type is mod 2 ** 32;
   type Mod_64_Type is mod 2 ** 64;
   type Mod_32_Array_Type is array (Index_Type'Range) of Mod_32_Type;

   Mod_32_Array : Mod_32_Array_Type := (others => Mod_32_Type'First);
   Index        : Index_Type        := Index_Type'First;
   First_Time   : Boolean           := True;

end Random_Normal;
```

Le fichier random_normal.adb

```
--                    M E R S E N N E   T W I S T E R
--                    ==============================
-- Pseudocode
-- The following generates uniformly 32 bit integers in the range [0, 2^32 - 1]
-- with the MT19937 algorithm:
--  // Create a length 624 array to store the state of the generator
--  var int[0..623] MT
--  var int y
--  // Initialise the generator from a seed
--  function initialiseGenerator ( 32-bit int seed ) {
--      MT[0] := seed
--      for i from 1 to 623 { // loop over each other element
--          MT[i] := last_32bits_of((69069 * MT[i-1]) + 1)
--      }
--  }
--
--  // Generate an array of 624 untempered numbers
--  function generateNumbers() {
--      for i from 0 to 622 {
--          y := 32nd_bit_of(MT[i]) + last_31bits_of(MT[i+1])
--          if y even {
--              MT[i] := MT[(i + 397) % 624] bitwise_xor (right_shift_by_1_bit(y))
--          } else if y odd {
--              MT[i] := MT[(i + 397) % 624] bitwise_xor (right_shift_by_1_bit(y))
--              bitwise_xor (2567483615)
--          }
--      }
--      y := 32nd_bit_of(MT[623]) + last_31bits_of(MT[0])
--      if y even {
--          MT[623] := MT[396] bitwise_xor (right_shift_by_1_bit(y))
--      } else if y odd {
--          MT[623] := MT[396] bitwise_xor (right_shift_by_1_bit(y))
--          bitwise_xor (2567483615)
--      }
--  }
--
--  // Extract a tempered pseudorandom number based on the i-th value
--  function extractNumber(int i) {
--      y := MT[i]
--      y := y bitwise_xor (right_shift_by_11_bits(y))
--      y := y bitwise_xor (left_shift_by_7_bits(y) bitwise_and (2636928640))
--      y := y bitwise_xor (left_shift_by_15_bits(y) bitwise_and (4022730752))
--      y := y bitwise_xor (right_shift_by_18_bits(y))
--      return y
--  }
--
--
-- Reference
-- M. Matsumoto and T. Nishimura, Mersenne twister: A 623-dimensionally
-- equidistributed uniform pseudorandom number generator,
-- ACM Trans. on Modeling and Computer Simulations, 1998.



with Ada.Exceptions;
with Ada.Numerics.Generic_Elementary_Functions;-- for sin log ......
with Ada.Text_Io;

package body Random_Normal
```

```
   is

   --|
   --| Initialise the generator from a seed
   --|
   procedure Initialize_Generator(Seed : in     Mod_32_Type := 5489)
      is
      --|
      --| needed to avoid 32 bits to overflow when *1812433253 ( see below)
      --|
      Mod_64 : Mod_64_Type;
      begin
      --|
      --| get start from seed
      --|
      Mod_32_Array(0) := Seed;--;and 16#FFFFFFFF#;
      --|
      --| loop to fill the array from seed ( done once at init)
      --|
      for I in 1 .. Mod_32_Array'Last loop
         --|
         --|no risk to overflow: changed to 64 bits storage before operation
         --|
         Mod_64 := Mod_64_Type(Mod_32_Array(I - 1)
                               xor Mod_32_Array(I - 1) / 2 ** 30) + Mod_64_Type(I);
         --|
         --|  here the risk is avoided 32 bits *1812433253 cannot overflow in 64 bit storage
         --|
         Mod_64 := 1812433253 * Mod_64;
         --|
         --| zero all high order bits keeping only 32 and back to 32 bit storage
         --|
         Mod_32_Array(I) := Mod_32_Type(Mod_64 and 16#FFFFFFFF#);
      end loop;
      --|
      --|                                        ========================
      --|                                        e r r o r   h a n d l e r
      --|                                        ========================
      --|
      exception

      when Programing_Error : others =>
      Ada.Text_Io.Put("Random_Normal.Initialize_Generator"
                      & Ada.Exceptions.Exception_Information(Programing_Error));
   end Initialize_Generator;

   procedure Generate_A_New_Array
      is
      -- Generate a new array of 624 untempered numbers from the previous state of the array
      Y : Mod_32_Type;
      begin
      --|
      --|loop
      --|
      for I in Index_Type'First .. (Index_Type'Last - 1) loop
         Y := (Mod_32_Array(I) and 16#80000000#) + (Mod_32_Array(I + 1) and 16#7FFFFFFF#);
         --|
         --| odd or even
         --|
         if (Y and 1) = 1
            then
            Mod_32_Array(I) := Mod_32_Array((I + 397) mod (Index_Type'Last + 1))
            xor Y / 2 xor 2567483615;
            else
            Mod_32_Array(I) := Mod_32_Array((I + 397) mod (Index_Type'Last + 1)) xor Y / 2;
         end if;
      end loop;
      --|
      --| last one
      --|
      Y := (Mod_32_Array(Index_Type'Last) and 16#80000000#) + (Mod_32_Array(0) and 16#7FFFFFFF#);
      --|
      --|odd or even
      --|
      if (Y and 1) = 1
         then
         Mod_32_Array(Index_Type'Last) := Mod_32_Array(Index_Type'Last) xor Y / 2 xor 2567483615;
         else
         Mod_32_Array(Index_Type'Last) := Mod_32_Array(Index_Type'Last) xor Y / 2;
      end if;
      --|
      --|                                        ========================
      --|                                        e r r o r   h a n d l e r
      --|                                        ========================
      --|
      exception

      when Programing_Error : others =>
      Ada.Text_Io.Put("Random_Normal.Generate_A_New_Array"
                      & Ada.Exceptions.Exception_Information(Programing_Error));
   end Generate_A_New_Array;
-- génération des nombre distribués uniformément sur 32 bits
   function Generate_Numbers
      return Mod_32_Type
      is
      Y : Mod_32_Type;
      begin
```

```
      --|
      --| init
      --|
      if First_Time
         then
         Initialize_Generator(Seed => 5489);
         First_Time := False;
         Index      := 0;
         Generate_A_New_Array;
      end if;
      --|
      --| see algo
      --|
      Y := Mod_32_Array(Index);
      Y := Y xor (Y / 2 ** 11);
      Y := Y xor (Y * 2 ** 7 and 2636928640);
      Y := Y xor (Y * 2 ** 15 and 4022730752);
      Y := Y xor (Y / 2 ** 18);
      --|
      --|all used up ? if so shuffle array, if not increment index for next call
      --|
      if Index = Index_Type'Last
         then
         Index := Index_Type'First;
         Generate_A_New_Array;
         else
         Index := 1 + Index;
      end if;
      --|
      --|debug
      --|
      --Ada.Text_Io.Put(" Index"
      --& Index_Type'Image(Index)
      --& " Y"
      --& Mod_32_Type'Image(Y));
      --|
      --|done
      --|
      return Y;
      --|
      --|                                    ========================
      --|                                    e r r o r   h a n d l e r
      --|                                    ========================
      --|
      exception

      when Programing_Error : others =>
      Ada.Text_Io.Put("Random_Normal.Generate_Numbers"
                      & Ada.Exceptions.Exception_Information(Programing_Error));
      return 0;
   end Generate_Numbers;
-- transformation en une distribution normale
   function Draw_Random(Mean               : in     Long_Float;
                        Standard_Deviation : in     Long_Float;
                        Max                : in     Long_Float := Long_Float'Last;
                        Min                : in     Long_Float := Long_Float'First)
      return Long_Float
      is

      function Get_One
         return Long_Float
         is
         --|
         --| needed to get log and ** on float
         --|
         package Math is new Ada.Numerics.Generic_Elementary_Functions(Float_Type => Long_Float);

         function Ln(X : in     Long_Float)
            return Long_Float
         renames Math.Log;

         function "**" (X : in Long_Float;
                        Y : in     Long_Float)
            return Long_Float
         renames Math."**";

         X             : Long_Float;
         Y             : Long_Float;
         Radius        : Long_Float;
         Normal_Random : Long_Float;

         begin
         --|
         --|box muller transform ( from a random 0..1 to normal distribution)
         --| z= x square root(-2Ln r/r)
         --|
         loop
            --|
            --| get two random in the range 0 .. 2**32
            --| and change range from 0 .. 2**32 to 0.0 .. 1.0
            --|
            Y := Long_Float(Generate_Numbers) / (2.0 ** 32 - 1.0);
            X := Long_Float(Generate_Numbers) / (2.0 ** 32 - 1.0);
            --|
            --|change to -1.0 .. 1.0
            --|
            X := X * 2.0 - 1.0;
```

```
            Y := Y * 2.0 - 1.0;
            --|
            --|check if within unit circle
            --|
            Radius := X ** 2 + Y ** 2;

            exit when Radius > 0.0 and Radius < 1.0;
         end loop;
         --|
         --| change into normal distribution with mean 0 and SD=1
         --|
         Normal_Random := X * (- 2.0 * Ln(Radius) / Radius) ** 0.5;
         --|
         --|change from normal with mean 0 and SD=1 into mean and sd as needed
         --|
         Normal_Random := Normal_Random * Standard_Deviation + Mean;
         --|
         --| done
         --|
         return Normal_Random;
         --|
         --|                                    ========================
         --|                                    e r r o r   h a n d l e r
         --|                                    ========================
         --|
         exception

            when Programing_Error : others =>
            Ada.Text_Io.Put("Random_Normal.Get_One"
                            & Ada.Exceptions.Exception_Information(Programing_Error));
            return 0.0;
      end Get_One;
      --|              | Draw_Random      |
      Result : Long_Float;
      begin
      --|
      --| check if within limits, if not try again
      --|
      loop
         Result := Get_One;
         --Ada.Text_Io.Put(  " Result"
         --& Long_Float'Image(Result));
         exit when Result in Min .. Max;
      end loop;
      --|
      --|done
      --|
      return Result;
      --|
      --|                                   ========================
      --|                                   e r r o r   h a n d l e r
      --|                                   ========================
      --|
      exception

      when Programing_Error : others =>
      Ada.Text_Io.Put("Random_Normal.Draw_Random"
                      & Ada.Exceptions.Exception_Information(Programing_Error));
      return 0.0;
   end Draw_Random;
end Random_Normal;
```

14 ANNEXES: LES CODES COMPLETS

14.1 LE PACKAGE READ_WRITE

Elle se compose de deux fichiers (ads et adb) et permet l'entrée sortie sans se préoccuper des détails

Fichier read_write.ads

```
with Ada.Text_Io;

package Read_Write
  is

  type String_Access_Type is access all String;

  --$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  --                  V A R I A B L E   R E A D
  --$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$


  procedure Read(Value  :    out Integer;
                 Header : in     String := "");

  procedure Read(Value  :    out Long_Integer;
                 Header : in     String := "");

  procedure Read(Value  :    out Float;
                 Header : in     String := "");

  procedure Read(Value  :    out Long_Float;
                 Header : in     String := "");

  procedure Read(Value  :    out Boolean;
                 Header : in     String := "");

  procedure Read(Value  :    out String;
                 Header : in     String := "");

  function Read(Header : in     String := "")
  return String_Access_Type;

  --$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  --                  V A R I A B L E   W R I T E
  --$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$



  procedure Write(Value  : in     Integer;
                  Header : in     String := "");

  procedure Write(Value  : in     Long_Integer;
                  Header : in     String := "");

  procedure Write(Value  : in     Float;
                  Header : in     String := "");

  procedure Write(Value  : in     Long_Float;
                  Header : in     String := "");

  procedure Write(Value  : in     Boolean;
                  Header : in     String := "");

  procedure Write(Value  : in     String;
                  Header : in     String := "");
```

```
  procedure Write_Modulo_Data(X                          : in     Integer;
                              Base                       : in     Integer;
                              Number_Of_Digits_On_Screen : Positive := 8);

  --$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
  --                  T E X T   F I L E   R E A D   &   W R I T E
  --$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

  type Access_To_File_Type is private;


  procedure Open_The_File_For_Read(File_Name      : in     String;
                                   Success        :    out Boolean;
                                   File_Reference :    out Access_To_File_Type);


  function Is_Reading_Over(File_Reference : in     Access_To_File_Type)
  return Boolean;


  function Read_A_Line(File_Reference : in     Access_To_File_Type)
  return String;


  function Read_A_Line(File_Reference : in     Access_To_File_Type)
  return String_Access_Type;


  procedure Open_The_File_For_Write(File_Name      : in     String;
                                    Success        :    out Boolean;
                                    File_Reference :    out Access_To_File_Type);


  procedure Append_A_Line(The_Line       : in     String;
                          File_Reference : in     Access_To_File_Type);


  procedure Close_The_Opened_File(File_Reference : in     Access_To_File_Type);

  private

  type Access_To_File_Type is access all Ada.Text_Io.File_Type;

end Read_Write;
```

Fichier read_write.adb

```
with Ada.Text_Io;
with Ada.Strings.Fixed;

--$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
--                  V A R I A B L E   R E A D
--$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
package body Read_Write
  is


  procedure Read(Value  :    out Integer;
                 Header : in     String := "")

     is
     Aborted_By_User : Boolean := True;
     begin
     loop
        --|
        --|       declare local temporary variables
        --|
        declare
        A_String    : String(1 .. 100) := (others => ' ');
        End_Of_Line : Natural          := 0;
        begin
           --|
           --|       put header
           --|
           Ada.Text_Io.Put(Item => Header);
           --|
           --|  remove all unused characters from input buffer
           --|
           Ada.Text_Io.Set_Col(File => Ada.Text_Io.Current_Input,
                               To   => 1);
           --|
           --|  get characters from user
           --|
           Ada.Text_Io.Get_Line(Item => A_String,
                                Last => End_Of_Line);
           --|
           --|  Cr means aborted by user
           --|
           Aborted_By_User := 0 = End_Of_Line;

           exit when Aborted_By_User;
           --|
           --| try translation of user's characters into integer
           --|
           Value := Integer'Value(A_String(A_String'First .. End_Of_Line));
```

```
         --|
         --| translation ok: exit loop
         --|
         exit;
         --|
         --| something's wrong in the translation: the program jumps here
         --|
         exception
         --|
         --| same treatment for all possible errors
         --|
         when others =>
         --|
         --| put error message and start again
         --|
         Ada.Text_Io.Put_Line(Item => " essayez encore ");
      end;
   end loop;
end Read;


procedure Read(Value  :    out Long_Integer;
               Header : in     String := "")
   is
   Aborted_By_User : Boolean := True;
   begin
   loop
      --|
      --|       declare local temporary variables
      --|
      declare
      A_String    : String(1 .. 100) := (others => ' ');
      End_Of_Line : Natural          := 0;
      begin
         --|
         --|       put header
         --|
         Ada.Text_Io.Put(Item => Header);
         --|
         --|  get characters from user
         --|
         Ada.Text_Io.Get_Line(Item => A_String,
                              Last => End_Of_Line);
         --|
         --|  Cr means aborted by user
         --|
         Aborted_By_User := 0 = End_Of_Line;

         exit when Aborted_By_User;
         --|
         --| try translation of user's characters into integer
         --|
         Value := Long_Integer'Value(A_String(A_String'First .. End_Of_Line));
         --|
         --| translation ok: exit loop
         --|
         exit;
         --|
         --| something's wrong in the translation: the program jumps here
         --|
         exception
         --|
         --| same treatment for all possible errors
         --|
         when others =>
         --|
         --| put error message and start again
         --|
         Ada.Text_Io.Put_Line(Item => " essayez encore ");
      end;
   end loop;
end Read;


procedure Read(Value  :    out Float;
               Header : in     String := "")
   is
   Aborted_By_User : Boolean := True;
   begin
   loop
      --|
      --|       declare local temporary variables
      --|
      declare
      A_String    : String(1 .. 100) := (others => ' ');
      End_Of_Line : Natural          := 0;
      begin
         --|
         --|       put header
         --|
         Ada.Text_Io.Put(Item => Header);
         --|
         --|  get characters from user
         --|
         Ada.Text_Io.Get_Line(Item => A_String,
                              Last => End_Of_Line);
         --|
```

```
         --|  Cr means aborted by user
         --|
         Aborted_By_User := 0 = End_Of_Line;

         exit when Aborted_By_User;
         --|
         --| try translation of user's characters into integer
         --|
         Value := Float'Value(A_String(A_String'First .. End_Of_Line));
         --|
         --| translation ok: exit loop
         --|
         exit;
         --|
         --| something's wrong in the translation: the program jumps here
         --|
         exception
         --|
         --| same treatment for all possible errors
         --|
         when others =>
         --|
         --| put error message and start again
         --|
         Ada.Text_Io.Put_Line(Item => " essayez encore ");
      end;
   end loop;
end Read;


procedure Read(Value  :    out Long_Float;
               Header : in     String := "")
   is
   begin
   loop
      --|
      --|       declare local temporary variables
      --|
      declare
      A_String    : String(1 .. 100) := (others => ' ');
      End_Of_Line : Natural          := 0;
      begin
         --|
         --|       put header
         --|
         Ada.Text_Io.Put(Item => Header);
         --|
         --|  get characters from user
         --|
         Ada.Text_Io.Get_Line(Item => A_String,
                              Last => End_Of_Line);
         --|
         --|  Cr means aborted by user
         --|
         exit when 0 = End_Of_Line;
         --|
         --| try translation of user's characters into integer
         --|
         Value := Long_Float'Value(A_String(A_String'First .. End_Of_Line));
         --|
         --| translation ok: exit loop
         --|
         exit;
         --|
         --| something's wrong in the translation: the program jumps here
         --|
         exception
         --|
         --| same treatment for all possible errors
         --|
         when others =>
         --|
         --| put error message and start again
         --|
         Ada.Text_Io.Put_Line(Item => " essayez encore ");
      end;
   end loop;
end Read;


procedure Read(Value  :    out Boolean;
               Header : in     String := "")
   is
   begin
   loop
      --|
      --|       declare local temporary variables
      --|
      declare
      A_String    : String(1 .. 100) := (others => ' ');
      End_Of_Line : Natural          := 0;
      begin
         --|
         --|       put header
         --|
         Ada.Text_Io.Put(Item => Header);
         --|
```

```
         --|  get characters from user
         --|
         Ada.Text_Io.Get_Line(Item => A_String,
                              Last => End_Of_Line);
         --|
         --|  Cr means aborted by user
         --|
         exit when 0 = End_Of_Line;
         --|
         --| try translation of user's characters into integer
         --|
         Value := Boolean'Value(A_String(A_String'First .. End_Of_Line));
         --|
         --| translation ok: exit loop
         --|
         exit;
         --|
         --| something's wrong in the translation: the program jumps here
         --|
         exception
         --|
         --| same treatment for all possible errors
         --|
         when others =>
         --|
         --| put error message and start again
         --|
         Ada.Text_Io.Put_Line(Item => " essayez encore ");
      end;
   end loop;
end Read;


procedure Read(Value  :    out String;
               Header : in     String := "")
   is
   procedure Insert(What : in     String;
                    Into : in out String)
      is
      L_What : constant Natural := Ada.Strings.Fixed.Index_Non_Blank(Source => What,
                                                                     Going  => Ada.Strings.Backward);

      L_Into : constant Natural := Integer'Max(Into'First - 1, Ada.Strings.Fixed.Index_Non_Blank(Source => Into,
                                                                                                Going  => Ada.Strings.Backward));
      begin
      Into(L_Into + 1 .. L_Into + 1 + L_What - What'First) := What(What'First .. L_What);
   end Insert;


   A_String    : String(1 .. 300) := (others => ' ');
   End_Of_Line : Natural          := 0;
   begin
   loop
      begin
         --|
         --|       put header
         --|
         Ada.Text_Io.Put(Item => Header);
         --|
         --|  get characters from user
         --|
         Ada.Text_Io.Get_Line(Item => A_String,
                              Last => End_Of_Line);
         --|
         --| clear
         --|
         Value := (others => ' ');
         --|
         --| insert input into output
         --|
         Insert(What => A_String(1 .. End_Of_Line),
                Into => Value);
         exit;
         --|
         --| something's wrong in the INPUT: the program jumps here
         --|
         exception
         --|
         --| same treatment for all possible errors mostly too many characters
         --|
         when others =>
         --|
         --| put error message and start again
         --|
         Ada.Text_Io.Put_Line(Item => " trop de car. essayez encore ");
      end;
   end loop;
end Read;


function Read(Header : in     String := "")
   return String_Access_Type
   is

   A_String    : String(1 .. 300) := (others => ' ');
   End_Of_Line : Natural          := 0;
   begin
```

```
   --|
   --|       put header
   --|
   Ada.Text_Io.Put(Item => Header);
   --|
   --|  get characters from user
   --|
   Ada.Text_Io.Get_Line(Item => A_String,
                        Last => End_Of_Line);

   return new String'(A_String(1 .. End_Of_Line));
end Read;

--$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
--                  V A R I A B L E   W R I T E
--$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$


procedure Write(Value  : in     Integer;
                Header : in     String := "")
   is
   begin
   Ada.Text_Io.Put_Line(Header & Integer'Image(Value));
end Write;


procedure Write(Value  : in     Long_Integer;
                Header : in     String := "")
   is
   begin
   Ada.Text_Io.Put_Line(Header & Long_Integer'Image(Value));
end Write;


procedure Write(Value  : in     Float;
                Header : in     String := "")
   is
   procedure To_Screen(Value : in     Float)
      is

      Strg          : String(1 .. 50);
      Last          : Natural := Strg'Last;
      First         : Natural := Strg'First;
      Decimal_Point : Natural := Strg'First;
      L             : Integer := 0;
      package F_Io is new Ada.Text_Io.Float_Io(Float);
      begin
      --|
      --|  to string
      --|
      F_Io.Put(To   => Strg,
               Item => Value,
               Aft  => 10,
               Exp  => 0);
      --|
      --|  remove non significant digits : 1.0000000007879  => 1.00000000000000
      --|
      for K in Strg'Range loop
         if Strg(K) in '1' .. '9'
            or L > 1
            then L := 1 + L;
            if L > Float'Digits
               then Strg(K) := '0';
            end if;
         end if;
      end loop;
      --|
      --|   get First non blank
      --|
      for I in Strg'Range loop
         if Strg(I) = ' '
            then
            First := I;
            else
            exit;
         end if;
      end loop;
      --|
      --|   decimal point position
      --|
      for I in Strg'Range loop
         if Strg(I) = '.'
            then
            Decimal_Point := I;
            exit;
         end if;
      end loop;
      --|
      --|   get last  non 0 or .
      --|
      if Decimal_Point /= Strg'First
         then
         for I in reverse Decimal_Point .. Strg'Last loop
            if Strg(I) = '0'
               or else Strg(I) = '.'
               then
```

```
               Last := I - 1;
               else
               exit;
            end if;
         end loop;
      end if;
      --|
      --|   output
      --|
      Ada.Text_Io.Put_Line(Header & Strg(First .. Last));
      --|
      --|   problem ??
      --|
      exception
      when others =>
      Ada.Text_Io.Put_Line(Header & Float'Image(Value));

   end To_Screen;
   begin
   if abs (Value) not in 1.0e-15 .. 1.0e15
      then
      Ada.Text_Io.Put_Line(Header & Float'Image(Value));
      else
      To_Screen(Value => Value);
   end if;
end Write;


procedure Write(Value  : in     Long_Float;
                Header : in     String := "")
   is
   procedure To_Screen(Value : in     Long_Float)
      is

      Strg          : String(1 .. 50);
      Last          : Natural := Strg'Last;
      First         : Natural := Strg'First;
      Decimal_Point : Natural := Strg'First;
      L             : Integer := 0;
      package F_Io is new Ada.Text_Io.Float_Io(Long_Float);
      begin
      --|
      --|  to string
      --|
      F_Io.Put(To   => Strg,
               Item => Value,
               Aft  => 10,
               Exp  => 0);
      --|
      --|  remove non significant digits
      --|
      for K in Strg'Range loop
         if Strg(K) in '1' .. '9'
            or L > 1
            then L := 1 + L;
            if L > Long_Float'Digits
               then Strg(K) := '0';
            end if;
         end if;
      end loop;
      --|
      --|   get First non blank
      --|
      for I in Strg'Range loop
         if Strg(I) = ' '
            then
            First := I;
            else
            exit;
         end if;
      end loop;
      --|
      --|   decimal point position
      --|
      for I in Strg'Range loop
         if Strg(I) = '.'
            then
            Decimal_Point := I;
            exit;
         end if;
      end loop;
      --|
      --|   get last  non 0 or .
      --|
      if Decimal_Point /= Strg'First
         then
         for I in reverse Decimal_Point .. Strg'Last loop
            if Strg(I) = '0'
               or else Strg(I) = '.'
               then
               Last := I - 1;
               else
               exit;
            end if;
         end loop;
      end if;
      --|
```

```
      --|   output
      --|
      Ada.Text_Io.Put_Line(Header & Strg(First .. Last));
      --|
      --|   problem ??
      --|
      exception
      when others =>
      Ada.Text_Io.Put(Header & Long_Float'Image(Value));

   end To_Screen;
   begin
   if abs (Value) not in 1.0e-15 .. 1.0e15
      then
      Ada.Text_Io.Put_Line(Header & Long_Float'Image(Value));
      else
      To_Screen(Value => Value);
   end if;
end Write;


procedure Write(Value  : in     Boolean;
                Header : in     String := "")
   is
   begin
   Ada.Text_Io.Put_Line(Header & Boolean'Image(Value));
end Write;


procedure Write(Value  : in     String;
                Header : in     String := "")
   is
   begin
   Ada.Text_Io.Put_Line(Header
                        & Ada.Strings.Fixed.Trim(Source => Value,
                                                 Side   => Ada.Strings.Right));
end Write;


procedure Write_Modulo_Data(X                         : in     Integer;
                            Base                      : in     Integer;
                            Number_Of_Digits_On_Screen : Positive := 8)
   is
   Mod_Length               : constant Integer := X'Size;-- number of   digits in x
   S                        : String(1 .. Mod_Length + 4);
   Rightmost_Sharp_Position : Integer;

   package Mod_32_Io is new Ada.Text_Io.Integer_Io(Integer);
   begin
   --|
   --| translate into string   base#xxxxxxxx#
   --|
   Mod_32_Io.Put(Item => X,
                 Base => Base,
                 To   => S);
   --|
   --| get position of last # on the right
   --|
   for I in reverse 1 .. Mod_Length + 3 loop
      if S(I) = '#'
         then
         Rightmost_Sharp_Position := I;
         exit;
      end if;
   end loop;
   --|
   --| put leading zeros
   --|
   for I in reverse 4 .. Rightmost_Sharp_Position loop
      S(I) := '0';
   end loop;
   --|
   --| to screen
   --|
   Ada.Text_Io.Put(S(Mod_Length + 4 - Number_Of_Digits_On_Screen .. Mod_Length + 3)
                   & ' ');
end Write_Modulo_Data;

--$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
--                  T E X T   F I L E   R E A D   &    W R I T E
--$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$



procedure Open_The_File_For_Read(File_Name      : in     String;
                                 Success        :    out Boolean;
                                 File_Reference :    out Access_To_File_Type)
   is
   begin

   File_Reference := new Ada.Text_Io.File_Type;

   Ada.Text_Io.Open(File => File_Reference.all,
                    Mode => Ada.Text_Io.In_File,
                    Name => File_Name);
   Success := True;
   exception
```

```
      when others =>
      Success := False;

   end Open_The_File_For_Read;


   function Is_Reading_Over(File_Reference : in     Access_To_File_Type)
      return Boolean
      is
      begin
      return Ada.Text_Io.End_Of_File(File => File_Reference.all);
   end Is_Reading_Over;


   function Read_A_Line(File_Reference : in     Access_To_File_Type)
      return String
      is
      The_Line              : String(1 .. 10000) := (others => ' ');
      Last_Character_In_Line : Natural           := 0;
      begin
      --|
      --| READ LINE FROM FILE
      --|
      Ada.Text_Io.Get_Line(File => File_Reference.all,
                           Item => The_Line,
                           Last => Last_Character_In_Line);

      return The_Line(1 .. Last_Character_In_Line);
   end Read_A_Line;


   function Read_A_Line(File_Reference : in     Access_To_File_Type)
      return String_Access_Type
      is
      The_Line              : String(1 .. 10000) := (others => ' ');
      Last_Character_In_Line : Natural           := 0;
      begin
      --|
      --| READ LINE FROM FILE
      --|
      Ada.Text_Io.Get_Line(File => File_Reference.all,
                           Item => The_Line,
                           Last => Last_Character_In_Line);

      return new String'(The_Line(1 .. Last_Character_In_Line));
   end Read_A_Line;


   procedure Open_The_File_For_Write(File_Name      : in     String;
                                     Success        :    out Boolean;
                                     File_Reference :    out Access_To_File_Type)
      is
      begin
      File_Reference := new Ada.Text_Io.File_Type;
      --|
      --|open the corresponding file for output
      --|
      Ada.Text_Io.Create(File => File_Reference.all,
                         Mode => Ada.Text_Io.Out_File,
                         Name => File_Name);
      Success := True;
      exception
      when others =>
      Success := False;

   end Open_The_File_For_Write;


   procedure Append_A_Line(The_Line       : in     String;
                           File_Reference : in     Access_To_File_Type)
      is
      begin
      --|
      --|put line
      --|
      Ada.Text_Io.Put_Line(File => File_Reference.all,
                           Item => The_Line);
   end Append_A_Line;


   procedure Close_The_Opened_File(File_Reference : in     Access_To_File_Type)
      is
      begin

      Ada.Text_Io.Close(File => File_Reference.all);
   end Close_The_Opened_File;


end Read_Write;
```

14.2 LE CODE DE LA TABLE DE HASHING EN TABLEAU SIMPLE

Fichier simple_hash.ads

```
generic

type Data_Type is private;

Null_Record         : Data_Type;
Record_Already_Used : Data_Type;

type Index_Type is mod <>;

Increment_After_Collision : Index_Type;

--|
--| for = operator
--|
with function "=" (Left  : in Data_Type;
                   Right : in     Data_Type) return Boolean is <>;


package Simple_Hash
  is
  Record_Not_Found_On_Delete             : exception;
  type Hash_Table_Type(Number_Of_Buckets : Index_Type := Index_Type'Last) is limited private;
  type Iterator_Type is limited private;

  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |                H_Of                 |
  --|                      |_____________________________________|
  --|                      |                                     |
  function H_Of(The_String : in     String)
  return Index_Type;
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |                Seek                 |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  procedure Seek(Data              : in     Data_Type;
                 In_The_Hash_Table : in     Hash_Table_Type;
                 Success           :    out Boolean;
                 Found_Record      : in out Data_Type;
                 Bucket_Number     : in     Index_Type);
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |                Add                  |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  procedure Add(Data              : in     Data_Type;
                To_The_Hash_Table : in out Hash_Table_Type;
                Bucket_Number     : in     Index_Type);
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |                Delete               |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  procedure Delete(Data              : in     Data_Type;
                   In_The_Hash_Table : in out Hash_Table_Type;
                   Bucket_Number     : in     Index_Type);
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |            Size_Of                  |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  function Size_Of(The_Hash_Table : in     Hash_Table_Type)
  return Integer;
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |             Initialize              |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  procedure Initialize(Iterator          :    out Iterator_Type;
                       For_The_Hash_Table : in out Hash_Table_Type);
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |            Is_Done                  |
  --|                      |_____________________________________|
```

```
   --|                     |                                |
   --|
   function Is_Done(Iterator : in     Iterator_Type)
   return Boolean;
   --|
   --|                     |________________________________|
   --|                     |                                |
   --|                     |      Current_Value             |
   --|                     |________________________________|
   --|                     |                                |
   --|
   function Current_Value(Iterator : in     Iterator_Type)
   return Data_Type;
   --|
   --|                     |________________________________|
   --|                     |                                |
   --|                     |      To_Next_Value             |
   --|                     |________________________________|
   --|                     |                                |
   --|
   procedure To_Next_Value(Iterator : in out Iterator_Type);
   --|                     |________________________________|
   --|                     |                                |
   --|                     |         Initialize             |
   --|                     |________________________________|
   --|                     |                                |
   --|
   procedure Initialize(Hash_Table : in out Hash_Table_Type);
   --|
   --|                     |________________________________|
   --|                     |                                |
   --|                     |       Print_Statistics         |
   --|                     |________________________________|
   --|                     |                                |
   --|
   procedure Print_Statistics(Hash_Table : in     Hash_Table_Type);

   private
   --|
   --|  hash table
   --|
   type Hash_Table_Array_Type is array (Index_Type range <>) of Data_Type;
   --|
   --|
   --|
   type Hash_Table_Type(Number_Of_Buckets : Index_Type := Index_Type'Last)
      is
      record
      Hash_Table_Array : aliased Hash_Table_Array_Type(0 .. Number_Of_Buckets) := (others => Null_Record);
      How_Many_Stored  : Natural                                               := 0;
   end record;
   --|
   --|  needed for the iterator
   --|
   type Hash_Table_Array_Access_Type is access all Hash_Table_Array_Type;
   --|
   --|  iterator
   --|
   type Iterator_Type is
      record
      Done                  : Boolean                     := False;
      Hash_Table_Index      : Index_Type                  := Index_Type'First;
      Number_Of_Buckets     : Index_Type                  := Index_Type'First;
      Hash_Table_Array_Access : Hash_Table_Array_Access_Type := null;
   end record;


   type Statistics_Type is
      record
      Collisions_Add    : Integer := 0;
      Collisions_Delete : Integer := 0;
      Collisions_Seek   : Integer := 0;
      Add_Calls         : Integer := 0;
      Delete_Calls      : Integer := 0;
      Seek_Calls        : Integer := 0;
   end record;
   Statistics : Statistics_Type;
end Simple_Hash;
```

Fichier simple_hash.adb

```
with Ada.Text_Io;
with Ada.Strings.Fixed;
use Ada.Strings.Fixed;


package body Simple_Hash
   is
   --|
   --|                     |________________________________|
   --|                     |                                |
   --|                     |            H_Of                |
   --|                     |________________________________|
   --|                     |                                |
```

```
--|

--
--
function H_Of(The_String : in     String)
   return Index_Type
   is
   --|
   --| From Compilers principles, techniques and tools
   --| AHO et all  addison wesley    page 436
   --|
   type H_Type is mod 2 ** 32;
   H : H_Type := H_Type'First;
   Z : H_Type := H_Type'First;
   begin
   for I in The_String'Range loop
      H := H * 2 ** 4;
      H := H + Character'Pos(The_String(I));
      if (H and 16#F0000000#) /= 0
         then
         Z := H;
         H := H / 2 ** 24;
         H := H xor Z;
         H := H and 16#0FFFFFFF#;
      end if;
   end loop;
   return Index_Type(H mod (1 + H_Type(Index_Type'Last)));
end H_Of;


--   Ajouter
--  Hash  <-  Calculer la valeur de hashing pour la clÃ©
--   boucle sur la taille du tableau
--    si Tableau(hash) = rien
--    ou alors  Tableau(hash) =  marqueur spÃ©cial
--    tableau(hash)<- Ã©lÃ©ment
--    sortie
--    fin si
--   hash <- (7+hash) modulo 13
--   fin boucle
--  erreur table pleine

--|
--|                     |_____________________________|
--|                     |                             |
--|                     |                Add          |
--|                     |_____________________________|
--|                     |                             |
--|
procedure Add(Data              : in     Data_Type;
              To_The_Hash_Table : in out Hash_Table_Type;
              Bucket_Number     : in     Index_Type)


   is
   Dummy_Bucket_Number : Index_Type := Bucket_Number;
   begin

   Statistics.Add_Calls := 1 + Statistics.Add_Calls;

   for K in To_The_Hash_Table.Hash_Table_Array'Range loop

      if To_The_Hash_Table.Hash_Table_Array(Dummy_Bucket_Number) = Null_Record
         or else To_The_Hash_Table.Hash_Table_Array(Dummy_Bucket_Number) = Record_Already_Used
         then
   To_The_Hash_Table.How_Many_Stored                     := To_The_Hash_Table.How_Many_Stored + 1;
         To_The_Hash_Table.Hash_Table_Array(Dummy_Bucket_Number) := Data;
         exit;
      end if;
      Statistics.Collisions_Add := 1 + Statistics.Collisions_Add;
      Dummy_Bucket_Number       := Dummy_Bucket_Number + Increment_After_Collision;
   end loop;
end Add;


--|
--|                     |_____________________________|
--|                     |                             |
--|                     |                Seek         |
--|                     |_____________________________|
--|                     |                             |
--|
procedure Seek(Data              : in     Data_Type;
               In_The_Hash_Table : in     Hash_Table_Type;
               Success           :    out Boolean;
               Found_Record      : in out Data_Type;
               Bucket_Number     : in     Index_Type)
   is
   Dummy_Bucket_Number : Index_Type := Bucket_Number;
   begin

   Statistics.Seek_Calls := 1 + Statistics.Seek_Calls;

   for K in In_The_Hash_Table.Hash_Table_Array'Range loop
      if In_The_Hash_Table.Hash_Table_Array(Dummy_Bucket_Number) = Data
         then
         Found_Record := In_The_Hash_Table.Hash_Table_Array(Dummy_Bucket_Number);
```

```
         Success      := True;
         return;
         elsif In_The_Hash_Table.Hash_Table_Array(Dummy_Bucket_Number) = Null_Record
         then
         Success := False;
         return;
      end if;
      Statistics.Collisions_Seek := 1 + Statistics.Collisions_Seek;
      Dummy_Bucket_Number        := Dummy_Bucket_Number + Increment_After_Collision;
   end loop;
   Success := False;
end Seek;



--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |                Delete               |
--|                     |_____________________________________|
--|                     |                                     |
--|
procedure Delete(Data              : in     Data_Type;
                 In_The_Hash_Table : in out Hash_Table_Type;
                 Bucket_Number     : in     Index_Type)
   is
   Dummy_Index : Index_Type := Bucket_Number;
   begin

   Statistics.Delete_Calls := 1 + Statistics.Delete_Calls;

   for K in In_The_Hash_Table.Hash_Table_Array'Range loop
      if In_The_Hash_Table.Hash_Table_Array(Dummy_Index) = Data
         then
         In_The_Hash_Table.Hash_Table_Array(Dummy_Index) := Record_Already_Used;
         In_The_Hash_Table.How_Many_Stored               := In_The_Hash_Table.How_Many_Stored - 1;
         return;
      end if;
      Statistics.Collisions_Delete := 1 + Statistics.Collisions_Delete;
      Dummy_Index                  := Dummy_Index + Increment_After_Collision;
   end loop;
   raise Record_Not_Found_On_Delete;
end Delete;
--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |            Size_Of                  |
--|                     |_____________________________________|
--|                     |                                     |
--|
function Size_Of(The_Hash_Table : in     Hash_Table_Type)
   return Integer
   is
   begin
   return The_Hash_Table.How_Many_Stored;
end Size_Of;
--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |             Initialize              |
--|                     |_____________________________________|
--|                     |                                     |
--|
procedure Initialize(Iterator           :    out Iterator_Type;
                     For_The_Hash_Table : in out Hash_Table_Type)
   is
   begin

   Iterator.Done                    := False;
   Iterator.Hash_Table_Index        := 0;
   Iterator.Number_Of_Buckets       := Index_Type'Last;
   Iterator.Hash_Table_Array_Access
   := new Hash_Table_Array_Type'(For_The_Hash_Table.Hash_Table_Array);

   for I in Index_Type'Range loop
      if Iterator.Hash_Table_Array_Access.all (I) /= Null_Record
         and Iterator.Hash_Table_Array_Access.all (I) /= Record_Already_Used
         then
         Iterator.Hash_Table_Index := I;
         return;
      end if;
   end loop;

   Iterator.Done := True;

end Initialize;
--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |            Is_Done                  |
--|                     |_____________________________________|
--|                     |                                     |
--|
function Is_Done(Iterator : in     Iterator_Type)
   return Boolean
   is
   begin
```

```
   return Iterator.Done;
end Is_Done;
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |      Current_Value                  |
--|                      |_____________________________________|
--|                      |                                     |
--|
function Current_Value(Iterator : in     Iterator_Type)
   return Data_Type
   is
   begin
   return Iterator.Hash_Table_Array_Access.all (Iterator.Hash_Table_Index);
end Current_Value;
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |      To_Next_Value                  |
--|                      |_____________________________________|
--|                      |                                     |
--|
procedure To_Next_Value(Iterator : in out Iterator_Type)
   is
   I : Index_Type := 1 + Iterator.Hash_Table_Index;
   begin

   while I in Iterator.Hash_Table_Index .. Index_Type'Last loop
      if Iterator.Hash_Table_Array_Access.all (I) /= Null_Record
         and Iterator.Hash_Table_Array_Access.all (I) /= Record_Already_Used
         then
         Iterator.Hash_Table_Index := I;
         return;
      end if;
      I := 1 + I;
   end loop;

   Iterator.Done := True;

end To_Next_Value;
--|                      |_____________________________________|
--|                      |                                     |
--|                      |         Initialize                  |
--|                      |_____________________________________|
--|                      |                                     |
--|
procedure Initialize(Hash_Table : in out Hash_Table_Type)
   is
   begin
   Hash_Table.Hash_Table_Array := (others => Null_Record);
   Hash_Table.How_Many_Stored  := 0;

   Statistics.Collisions_Add    := 0;
   Statistics.Collisions_Delete := 0;
   Statistics.Collisions_Seek   := 0;
   Statistics.Add_Calls         := 0;
   Statistics.Delete_Calls      := 0;
   Statistics.Seek_Calls        := 0;

end Initialize;
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |       Print_Statistics              |
--|                      |_____________________________________|
--|                      |                                     |
--|
procedure Print_Statistics(Hash_Table : in     Hash_Table_Type)
   is
   package My_Float_Io is new Ada.Text_Io.Float_Io(Float);
   begin
   Ada.Text_Io.Put_Line("How_Many_Stored" & Integer'Image(Hash_Table.How_Many_Stored)
                        & " Collisions_Add" & Integer'Image(Statistics.Collisions_Add)
                        & " Collisions_Delete" & Integer'Image(Statistics.Collisions_Delete)
                        & " Collisions_Seek" & Integer'Image(Statistics.Collisions_Seek)
                        & " Fill ratio "
                        & Integer'Image(100 * Hash_Table.How_Many_Stored
                                        / Integer(Hash_Table.Number_Of_Buckets))
                        & "%");

   Ada.Text_Io.Put_Line(" Add_Calls" & Integer'Image(Statistics.Add_Calls)
                        & " Delete_Calls" & Integer'Image(Statistics.Delete_Calls)
                        & " Seek_Calls" & Integer'Image(Statistics.Seek_Calls));


   Ada.Text_Io.Put_Line("Average collisions per ADD call  ");
   My_Float_Io.Put(Item => Float(Statistics.Collisions_Add) / Float(Statistics.Add_Calls),
                   Fore => 2,
                   Aft  => 2,
                   Exp  => 0);
   Ada.Text_Io.New_Line;

   Ada.Text_Io.Put_Line("Average collisions per DELETE call  ");
   My_Float_Io.Put(Item => Float(Statistics.Collisions_Delete) / Float(Statistics.Delete_Calls),
                   Fore => 2,
                   Aft  => 2,
                   Exp  => 0);
```

```
      Ada.Text_Io.New_Line;


      Ada.Text_Io.Put_Line("Average collisions per SEEK call ");
      My_Float_Io.Put(Item => Float(Statistics.Collisions_Seek) / Float(Statistics.Seek_Calls),
                      Fore => 2,
                      Aft  => 2,
                      Exp  => 0);

   end Print_Statistics;

end Simple_Hash;
```

14.3 LE CODE DE LA TABLE DE HASHING EN TABLEAU CHAÎNÉ

Fichier linked_hash.ads

```
with Ada.Unchecked_Deallocation;
--|
--|
--|  needed for hash_of availaility
--|
generic

type Data_Type is private;


--|
--| for = operator
--|
with function "=" (Left  : in Data_Type;
                   Right : in     Data_Type)
  return Boolean
is <>;

package Linked_Hash
  is
  Record_Not_Found_On_Delete                    : exception;
  type Linked_Hash_Table_Type(Number_Of_Buckets : Positive) is limited private;
  type Iterator_Type is limited private;

  --|
  --|                      |_____________________________|
  --|                      |                             |
  --|                      |                H_Of         |
  --|                      |_____________________________|
  --|                      |                             |
  --|
  function H_Of(The_String : in     String)
  return Integer;
  --|
  --|                      |_____________________________|
  --|                      |                             |
  --|                      |                Seek         |
  --|                      |_____________________________|
  --|                      |                             |
  --|
  procedure Seek(Data              : in     Data_Type;
                 In_The_Hash_Table : in     Linked_Hash_Table_Type;
                 Success           :    out Boolean;
                 Found_Record      : in out Data_Type;
                 Bucket_Number     : in     Natural);
  --|
  --|                      |_____________________________|
  --|                      |                             |
  --|                      |                Add          |
  --|                      |_____________________________|
  --|                      |                             |
  --|
  procedure Add(Data              : in     Data_Type;
                To_The_Hash_Table : in out Linked_Hash_Table_Type;
                Bucket_Number     : in     Natural);
  --|
  --|                      |_____________________________|
  --|                      |                             |
  --|                      |                Delete       |
  --|                      |_____________________________|
  --|                      |                             |
  --|
  procedure Delete(Data              : in     Data_Type;
                   In_The_Hash_Table : in out Linked_Hash_Table_Type;
                   Bucket_Number     : in     Natural);
  --|
  --|                      |_____________________________|
  --|                      |                             |
  --|                      |            Size_Of          |
  --|                      |_____________________________|
  --|                      |                             |
  --|
```

```
function Size_Of(The_Hash_Table : in     Linked_Hash_Table_Type)
return Integer;
--|
--|                       |_____________________________________|
--|                       |                                     |
--|                       |             Initialize              |
--|                       |_____________________________________|
--|                       |                                     |
--|
procedure Initialize(Iterator           :    out Iterator_Type;
                     For_The_Hash_Table : in out Linked_Hash_Table_Type);
--|
--|                       |_____________________________________|
--|                       |                                     |
--|                       |            Is_Done                  |
--|                       |_____________________________________|
--|                       |                                     |
--|
function Is_Done(Iterator : in     Iterator_Type)
return Boolean;
--|
--|                       |_____________________________________|
--|                       |                                     |
--|                       |      Current_Value                  |
--|                       |_____________________________________|
--|                       |                                     |
--|
function Current_Value(Iterator : in     Iterator_Type)
return Data_Type;
--|
--|                       |_____________________________________|
--|                       |                                     |
--|                       |      To_Next_Value                  |
--|                       |_____________________________________|
--|                       |                                     |
--|
procedure To_Next_Value(Iterator : in out Iterator_Type);
--|                       |_____________________________________|
--|                       |                                     |
--|                       |         Initialize                  |
--|                       |_____________________________________|
--|                       |                                     |
--|
procedure Initialize(Hash_Table : in out Linked_Hash_Table_Type);
--|
--|                       |_____________________________________|
--|                       |                                     |
--|                       |       Print_Statistics              |
--|                       |_____________________________________|
--|                       |                                     |
--|
procedure Print_Statistics(Hash_Table : in     Linked_Hash_Table_Type);


private

type Node_Type;
--|
--|
--|
type Node_Access_Type is access all Node_Type;
--|
--|
--|
type Node_Type is
   record
   Data           : Data_Type;
   Access_To_Next : Node_Access_Type := null;
end record;
--|
--|  used in delete
--|
procedure Free_Node is new Ada.Unchecked_Deallocation(Object => Node_Type,
                                                      Name   => Node_Access_Type);
--|
--|  hash table
--|
type Hash_Table_Array_Type is array (Integer range <>) of Node_Access_Type;
--|
--|
--|
type Linked_Hash_Table_Type(Number_Of_Buckets : Positive)
   is
   record
   Hash_Table_Array : aliased Hash_Table_Array_Type(0 .. Number_Of_Buckets) := (others => null);
   How_Many_Stored  : Natural                                               := 0;
end record;
--|
--|  needed for the iterator
--|
type Hash_Table_Array_Access_Type is access all Hash_Table_Array_Type;
--|
--|  iterator
--|
type Iterator_Type is
   record
```

```
      Current_Node_Access     : Node_Access_Type            := null;
      Hash_Table_Index        : Integer                     := 0;
      Number_Of_Buckets       : Integer                     := 0;
      Hash_Table_Array_Access : Hash_Table_Array_Access_Type := null;
   end record;

   type Statistics_Type is
      record
      Add_Calls    : Integer := 0;
      Delete_Calls : Integer := 0;
      Seek_Calls   : Integer := 0;

   end record;
   Statistics : Statistics_Type;
end Linked_Hash;
```

Fichier linked_hash.adb

```
--|
--|
--|
--|                  E X A M P L E   O F  H A S H   T A B L E   S T R U C T U R E
--|
--|
--|
--|
--|
--|       |------------- BUCKETS NUMBERS
--|       |
--|       V
--|
--|    |    |    |----------------------------|    |----------------------------|    |----------------------------|
--|    | 1  |--->|HASH RECORD | Access_To_Next |--->|HASH RECORD | Access_To_Next|--->|HASH RECORD | Access_To_Next|---> NULL
--|    |    |    |____________________________|    |____________________________|    |____________________________|
--|    |    |    |
--|    |  2 |--> NULL
--|    |    |
--|    |    |    |----------------------------|
--|    | 3  |--->|HASH RECORD | Access_To_Next |---> NULL
--|    |    |    |____________________________|
--|    |    |
--|    |    |    |----------------------------|
--|    | 4  |--->|HASH RECORD | Access_To_Next |--->
--|    |    |    |____________________________|
--|    |    |
--|    |    |    |----------------------------|    |----------------------------|
--|    | 5  |--->|HASH RECORD | Access_To_Next |--->|HASH RECORD | Access_To_Next |---> NULL
--|    |    |    |____________________________|    |____________________________|
--|    |    |    |
--|    |    |    |
--|    | 6  |---> NULL
--|    |    |    |
--|    |    |    |
--|    |  7 |--->NULL
--|    |    |    |
--|    |    |    |
--|
--|
```

```
with Ada.Text_Io;

package body Linked_Hash
   is
   --|
   --|                     |_____________________________________|
   --|                     |                                     |
   --|                     |                H_Of                 |
   --|                     |_____________________________________|
   --|                     |                                     |
   function H_Of(The_String : in     String)
      return Integer
      is
      --|
      --| 32 bits avec fonctions logiques   or and xor ,.....
      --|
      type H_Type is mod 2 ** 32;
      --|
      --| H <- 0   z <- 0
      --|
      H : H_Type := H_Type'First;
      Z : H_Type := H_Type'First;

      procedure Debug_Hexadecimal(X : in     H_Type)
         is
         package Iio is new Ada.Text_Io.Modular_Io(H_Type);
```

```
      begin
      Iio.Put(Item  => X,
              Base  => 16,
              Width => 12);
      Ada.Text_Io.Put(' ');
   end Debug_Hexadecimal;

   begin
   --Ada.Text_Io.Put('!'&The_String&'!');
   --|
   --| Boucle sur tous les caract�es
   --|
   for I in The_String'Range loop
      --|
      --| D�aler H de 4 bits vers la gauche
      --|
      H := H * 2 ** 4;
      --|
      --|H <- H + valeur ASCII du caract�e courant
      --|
      H := H + Character'Pos(The_String(I));
      --|
      --|Si l'un des 4 bits de poids le plus �ev�est �al �1
      --|
      if (H and 16#F0000000#) /= 0
         --|
         --|Alors
         --|
         then
         --|
         --|Z <- H
         --|
         Z := H;
         --Ada.Text_Io.Put("Z ");
         --Debug_Hexadecimal(X => Z);
         --|
         --|D�aler h de 24 bit ( 6 positions hexadecimal) vers la droite
         --|
         H := H / 2 ** 24;
         --Ada.Text_Io.Put("h24 ");
         --Debug_Hexadecimal(X => H);
         --|
         --|H <- H   xor   Z
         --|
         H := H xor Z;
         --Ada.Text_Io.Put("h xor ");
         --Debug_Hexadecimal(X => H);
         --|
         --|Mettre �0 les 4 bits de poids le plus �ev�de H
         --|
         H := H and 16#0FFFFFFF#;
         --Ada.Text_Io.Put(" and  ");
         --Debug_Hexadecimal(X => H);
         --Ada.Text_Io.New_Line;
         --|
         --|Fin si
         --|
      end if;
   end loop;
   --|
   --|Renvoi H
   return Integer(H);
end H_Of;
--|
--|                     |_____________________________|
--|                     |                             |
--|                     |            Seek             |
--|                     |_____________________________|
--|                     |                             |
--|
procedure Seek(Data              : in     Data_Type;
               In_The_Hash_Table : in     Linked_Hash_Table_Type;
               Success           :    out Boolean;
               Found_Record      : in out Data_Type;
               Bucket_Number     : in     Natural)
   is
   Local_Bucket_Number : Natural          := Bucket_Number mod (1 + In_The_Hash_Table.Number_Of_Buckets);
   Node_Access         : Node_Access_Type := In_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number);
   begin
   Statistics.Seek_Calls := 1 + Statistics.Seek_Calls;
   Success               := False;
   while null /= Node_Access loop
      if Data = Node_Access.all.Data
         then
         Success      := True;
         Found_Record := Node_Access.all.Data;
      end if;
      Node_Access := Node_Access.Access_To_Next;
   end loop;
end Seek;
--|
--|                     |_____________________________|
--|                     |                             |
--|                     |            Add              |
--|                     |_____________________________|
--|                     |                             |
--|
```

```
procedure Add(Data              : in     Data_Type;
              To_The_Hash_Table : in out Linked_Hash_Table_Type;
              Bucket_Number     : in     Natural)
   is
   Local_Bucket_Number : Natural := Bucket_Number mod (1 + To_The_Hash_Table.Number_Of_Buckets);
   begin
   Statistics.Add_Calls := 1 + Statistics.Add_Calls;
   --|
   --| insert
   --|
   To_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number)
   := new Node_Type'(Data           => Data,
                     Access_To_Next => To_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number));

   To_The_Hash_Table.How_Many_Stored := 1 + To_The_Hash_Table.How_Many_Stored;
end Add;
--|
--|                      |_____________________________|
--|                      |                             |
--|                      |           Delete            |
--|                      |_____________________________|
--|                      |                             |
--|
procedure Delete(Data              : in     Data_Type;
                 In_The_Hash_Table : in out Linked_Hash_Table_Type;
                 Bucket_Number     : in     Natural)
   is
   Local_Bucket_Number: Natural  := Bucket_Number mod (1 + In_The_Hash_Table.Number_Of_Buckets);
   Current_Node_Access:Node_Access_Type := In_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number);
 Previous_Node_Access: Node_Access_Type := In_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number);

   begin
   Statistics.Delete_Calls := 1 + Statistics.Delete_Calls;
   --| find the item to remove
   while null /= Current_Node_Access loop
      exit when Data = Current_Node_Access.all.Data;
      Previous_Node_Access := Current_Node_Access;
      Current_Node_Access  := Current_Node_Access.Access_To_Next;
   end loop;

   if null = Current_Node_Access
      then
      raise Record_Not_Found_On_Delete;
   end if;

   --| first one on linked list
   if Current_Node_Access = Previous_Node_Access
      then
      In_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number)
      := In_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number).Access_To_Next;
      else
      Previous_Node_Access.Access_To_Next := Current_Node_Access.Access_To_Next;
   end if;
   Free_Node(Current_Node_Access);
   In_The_Hash_Table.How_Many_Stored := In_The_Hash_Table.How_Many_Stored - 1;
end Delete;
--|
--|                      |_____________________________|
--|                      |                             |
--|                      |           Size_Of           |
--|                      |_____________________________|
--|                      |                             |
--|
function Size_Of(The_Hash_Table : in     Linked_Hash_Table_Type)
   return Integer
   is
   begin
   return The_Hash_Table.How_Many_Stored;
end Size_Of;
--|
--|                      |_____________________________|
--|                      |                             |
--|                      |           Is_Done           |
--|                      |_____________________________|
--|                      |                             |
--|
function Is_Done(Iterator : in     Iterator_Type)
   return Boolean
   is
   begin
   return Iterator.Hash_Table_Index > Iterator.Number_Of_Buckets;
end Is_Done;
--|
--|                      |_____________________________|
--|                      |                             |
--|                      |      Current_Value          |
--|                      |_____________________________|
--|                      |                             |
--|
function Current_Value(Iterator : in     Iterator_Type)
   return Data_Type
   is
   begin
   return Iterator.Current_Node_Access.Data;
end Current_Value;
--|
--|                      |_____________________________|
```

```
--|                     |                                  |
--|                     |      To_Next_Value               |
--|                     |__________________________________|
--|                     |                                  |
--|
procedure Get_Next_Valid_Record(Iterator : in out Iterator_Type)
   is
   begin

   loop
      --Ada.Text_Io.Put_Line("  Iterator.Hash_Table_Index" & Integer'Image(Iterator.Hash_Table_Index));

      exit when Iterator.Current_Node_Access /= null;
      Iterator.Hash_Table_Index := 1 + Iterator.Hash_Table_Index;
      exit when Iterator.Hash_Table_Index > Iterator.Number_Of_Buckets;
      Iterator.Current_Node_Access := Iterator.Hash_Table_Array_Access.all (Iterator.Hash_Table_Index);

   end loop;

end Get_Next_Valid_Record;
--|                     |__________________________________|
--|                     |                                  |
--|                     |      To_Next_Value               |
--|                     |__________________________________|
--|                     |                                  |
--|
procedure To_Next_Value(Iterator : in out Iterator_Type)
   is
   begin
   -- go  one step further
   Iterator.Current_Node_Access := Iterator.Current_Node_Access.Access_To_Next;
   -- get next valid record (if any)
   Get_Next_Valid_Record(Iterator => Iterator);
end To_Next_Value;
--|
--|                     |__________________________________|
--|                     |                                  |
--|                     |            Initialize            |
--|                     |__________________________________|
--|                     |                                  |
--|
procedure Initialize(Iterator           :    out Iterator_Type;
                     For_The_Hash_Table : in out Linked_Hash_Table_Type)
   is
   begin
   --    iterator can be used several times so init everything every time
   Iterator.Hash_Table_Index        := 0;
   Iterator.Current_Node_Access     := For_The_Hash_Table.Hash_Table_Array(Iterator.Hash_Table_Index);
   Iterator.Number_Of_Buckets       := For_The_Hash_Table.Number_Of_Buckets;
   Iterator.Hash_Table_Array_Access := new Hash_Table_Array_Type'(For_The_Hash_Table.Hash_Table_Array);
   -- seek  first non empty position in hash table  (if any)
   Get_Next_Valid_Record(Iterator => Iterator);

end Initialize;
--|
--|                     |__________________________________|
--|                     |                                  |
--|                     |         Initialize               |
--|                     |__________________________________|
--|                     |                                  |
--|
procedure Initialize(Hash_Table : in out Linked_Hash_Table_Type)
   is
   Dummy_Node_Access : Node_Access_Type;
   begin

   for Local_Bucket_Number in Hash_Table.Hash_Table_Array'Range loop
      while null /= Hash_Table.Hash_Table_Array(Local_Bucket_Number) loop

         Dummy_Node_Access := Hash_Table.Hash_Table_Array(Local_Bucket_Number);

         Hash_Table.Hash_Table_Array(Local_Bucket_Number) := Dummy_Node_Access.Access_To_Next;

         Free_Node(Dummy_Node_Access);
      end loop;
   end loop;
   Hash_Table.How_Many_Stored := 0;


   Statistics.Add_Calls    := 0;
   Statistics.Delete_Calls := 0;
   Statistics.Seek_Calls   := 0;
end Initialize;
--|
--|                     |__________________________________|
--|                     |                                  |
--|                     |       Print_Statistics           |
--|                     |__________________________________|
--|                     |                                  |
--|
procedure Print_Statistics(Hash_Table : in     Linked_Hash_Table_Type)
   is
   --package My_Float_Io is new Ada.Text_Io.Float_Io(Float);
   begin
   Ada.Text_Io.Put_Line("How_Many_Stored" & Integer'Image(Hash_Table.How_Many_Stored));

   Ada.Text_Io.Put_Line(" Add_Calls" & Integer'Image(Statistics.Add_Calls)
```

```
                        & " Delete_Calls" & Integer'Image(Statistics.Delete_Calls)
                        & " Seek_Calls" & Integer'Image(Statistics.Seek_Calls));
  end Print_Statistics;
end Linked_Hash;
```

14.4 LE CODE DU STACK (TABLEAU)

Fichier .ads :

```
generic

type Data_Type is private;
Stack_Size : Positive;


package Simple_Stack
  is

  Stack_Full  : exception;
  Stack_Empty : exception;


  type Iterator_Type is limited private;
  type Stack_Type is limited private;

  --|
  --|
  ------------------------------------------------------------------
  ------------------------------------------------------------------
  ------------------   S T A C K   O P E R A T I O N S   ----------
  ------------------------------------------------------------------
  ------------------------------------------------------------------
  --|
  --|

  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |                push                 |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  procedure Push(Data         : in     Data_Type;
                 In_The_Stack : in out Stack_Type);
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |                pop                  |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  procedure Pop(The_Stack : in out Stack_Type);
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |            Size_Of                  |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  function Size_Of(The_Stack : in     Stack_Type)
  return Integer;
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |            top_Of                   |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  function Top_Of(The_Stack : in     Stack_Type)
  return Data_Type;
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |         Initialize                  |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  procedure Initialize(The_Stack : in out Stack_Type);
  --|
  --|
  ------------------------------------------------------------------
  ------------------------------------------------------------------
  -------------------          I T E R A T O R     ------------------
  ------------------------------------------------------------------
```

```
------------------------------------------------------------------
--|
--|

--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |             Initialize              |
--|                      |_____________________________________|
--|                      |                                     |
--|
procedure Initialize(Iterator      :    out Iterator_Type;
                     For_The_Stack : in     Stack_Type);
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |              Is_Done                |
--|                      |_____________________________________|
--|                      |                                     |
--|
function Is_Done(Iterator : in     Iterator_Type)
return Boolean;
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |       Current_Value                 |
--|                      |_____________________________________|
--|                      |                                     |
--|
function Current_Value(Iterator : in     Iterator_Type)
return Data_Type;
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |       To_Next_Value                 |
--|                      |_____________________________________|
--|                      |                                     |
--|
procedure To_Next_Value(Iterator : in out Iterator_Type);
--|
--|
------------------------------------------------------------------
------------------------------------------------------------------
-------------------   S T A T I S T I C  S    --------------------
------------------------------------------------------------------
------------------------------------------------------------------
--|
--|

--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |        Print_Statistics             |
--|                      |_____________________________________|
--|                      |                                     |
--|
procedure Print_Statistics(Stack : in     Stack_Type);
--|
--|
------------------------------------------------------------------
------------------------------------------------------------------
-------------------  P R I V A T E   P A R T   --------------------
------------------------------------------------------------------
------------------------------------------------------------------
--|
--|
private
--|
--|  stack table
--|
type Stack_Table_Array_Type is array (Positive range <>) of Data_Type;
--|
--|
--|
type Stack_Type(Size : Positive := Stack_Size)
   is
   record
   Stack_Table_Array : aliased Stack_Table_Array_Type(1 .. Stack_Size);
   Current_Index     : Positive := 1;
end record;
--|
--|  needed for the iterator
--|
type Stack_Table_Array_Access_Type is access all Stack_Table_Array_Type;
--|
--|  iterator
--|
type Iterator_Type is
   record
   Stack_Table_Index        : Integer                       := 0;
   Stack_Table_Array_Access : Stack_Table_Array_Access_Type := null;
end record;


type Statistics_Type is
   record
   Collisions_Add    : Integer := 0;
```

```
      Collisions_Delete : Integer := 0;
      Collisions_Seek   : Integer := 0;
      Add_Calls         : Integer := 0;
      Delete_Calls      : Integer := 0;
      Seek_Calls        : Integer := 0;
   end record;

   Statistics : Statistics_Type;

end Simple_Stack;
```

Fichier .adb :

```
package body Simple_Stack
   is
   --|
   --|
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   -------------------   S T A C K   O P E R A T I O N S   ----------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   --|
   --|

   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |                push                 |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   procedure Push(Data         : in     Data_Type;
                  In_The_Stack : in out Stack_Type)
      is
      begin
      In_The_Stack.Stack_Table_Array(In_The_Stack.Current_Index) := Data;
      In_The_Stack.Current_Index
      := 1 + In_The_Stack.Current_Index;
      exception
      when Constraint_Error => raise Stack_Full;
   end Push;
   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |                pop                  |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   procedure Pop(The_Stack : in out Stack_Type)
      is
      begin
      The_Stack.Current_Index := The_Stack.Current_Index - 1;

      exception
      when Constraint_Error => raise Stack_Empty;
   end Pop;
   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |             Size_Of                 |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   function Size_Of(The_Stack : in     Stack_Type)
      return Integer
      is
      begin
      return The_Stack.Current_Index - 1;
   end Size_Of;
   --|
   --|             Stack_Table_Array_Access => new Stack_Table_Array_Type'(For_The_Stack.Stack_Table_Array));

   end Initialize;
   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |             Is_Done                 |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   function Is_Done(Iterator : in     Iterator_Type)
      return Boolean
      is
      begin
      return Iterator.Stack_Table_Index = 0;--Iterator.Last_Index;
   end Is_Done;
   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |       Current_Value                 |
   --|                      |_____________________________________|
   --|                      |                                     |
```

```
   --|
   function Current_Value(Iterator : in     Iterator_Type)
      return Data_Type
      is
      begin
      return Iterator.Stack_Table_Array_Access.all (Iterator.Stack_Table_Index);
   end Current_Value;
   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |      To_Next_Value                  |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   procedure To_Next_Value(Iterator : in out Iterator_Type)
      is
      begin
      Iterator.Stack_Table_Index := Iterator.Stack_Table_Index - 1;
   end To_Next_Value;
   --|
   --|
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   -------------------   S T A T I S T I C  S    --------------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   --|
   --|

   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |       Print_Statistics              |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   procedure Print_Statistics(Stack : in     Stack_Type)
      is
      begin
      null;
   end Print_Statistics;

end Simple_Stack;               |_____________________________________|
   --|                      |                                     |
   --|                      |            top_Of                   |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   function Top_Of(The_Stack : in     Stack_Type)
      return Data_Type
      is
      begin
      return The_Stack.Stack_Table_Array(The_Stack.Current_Index - 1);

      exception
      when Constraint_Error => raise Stack_Empty;
   end Top_Of;
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |         Initialize                  |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   procedure Initialize(The_Stack : in out Stack_Type)
      is
      begin
      The_Stack.Current_Index := 1;
   end Initialize;
   --|
   --|
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   ------------------  A C T I V E     I T E R A T O R   ------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   --|
   --|

   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |             Initialize              |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   procedure Initialize(Iterator      :    out Iterator_Type;
                        For_The_Stack : in     Stack_Type)
      is

      begin
      Iterator := (Stack_Table_Index        => For_The_Stack.Current_Index - 1,
                   Stack_Table_Array_Access => new Stack_Table_Array_Type'(For_The_Stack.Stack_Table_Array));

   end Initialize;
   --|
   --|                      |_____________________________________|
   --|                      |                                     |
```

```
   --|                     |           Is_Done            |
   --|                     |______________________________|
   --|                     |                              |
   --|
   function Is_Done(Iterator : in     Iterator_Type)
      return Boolean
      is
      begin
      return Iterator.Stack_Table_Index = 0;--Iterator.Last_Index;
   end Is_Done;
   --|
   --|                     |______________________________|
   --|                     |                              |
   --|                     |      Current_Value           |
   --|                     |______________________________|
   --|                     |                              |
   --|
   function Current_Value(Iterator : in     Iterator_Type)
      return Data_Type
      is
      begin
      return Iterator.Stack_Table_Array_Access.all (Iterator.Stack_Table_Index);
   end Current_Value;
   --|
   --|                     |______________________________|
   --|                     |                              |
   --|                     |      To_Next_Value           |
   --|                     |______________________________|
   --|                     |                              |
   --|
   procedure To_Next_Value(Iterator : in out Iterator_Type)
      is
      begin
      Iterator.Stack_Table_Index := Iterator.Stack_Table_Index - 1;
   end To_Next_Value;
   --|
   --|
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   ------------------   S T A T I S T I C  S    ---------------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   --|
   --|

   --|
   --|                     |______________________________|
   --|                     |                              |
   --|                     |       Print_Statistics       |
   --|                     |______________________________|
   --|                     |                              |
   --|
   procedure Print_Statistics(Stack : in     Stack_Type)
      is
      begin
      null;
   end Print_Statistics;

end Simple_Stack;
```

14.5 LE CODE DU STACK (POINTEURS)

Fichier .ads :

```
with Ada.Unchecked_Deallocation;
--|
--|
--|
--|
generic

type Data_Type is private;


package Linked_Stack
  is


  Stack_Empty : exception;

  type Iterator_Type is limited private;
  type Stack_Type is limited private;

  --|
  --|
  ------------------------------------------------------------------
  ------------------------------------------------------------------
  ------------------   S T A C K   O P E R A T I O N S    ----------
  ------------------------------------------------------------------
  ------------------------------------------------------------------
```

```
--|
--|

--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |                push                 |
--|                     |_____________________________________|
--|                     |                                     |
--|
procedure Push(Data         : in     Data_Type;
               In_The_Stack : in out Stack_Type);
--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |                pop                  |
--|                     |_____________________________________|
--|                     |                                     |
--|
procedure Pop(The_Stack : in out Stack_Type);
--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |            Size_Of                  |
--|                     |_____________________________________|
--|                     |                                     |
--|
function Size_Of(The_Stack : in     Stack_Type)
return Integer;
--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |            top_Of                  |
--|                     |_____________________________________|
--|                     |                                     |
--|
function Top_Of(The_Stack : in     Stack_Type)
return Data_Type;
--|                     |_____________________________________|
--|                     |                                     |
--|                     |         Initialize                  |
--|                     |_____________________________________|
--|                     |                                     |
--|
procedure Initialize(The_Stack : in out Stack_Type);
--|
--|
-------------------------------------------------------------------
-------------------------------------------------------------------
------------------          I T E R A T O R     ------------------
-------------------------------------------------------------------
-------------------------_____________________________________|
--|                     |                                     |
--|
procedure Print_Statistics(Stack : in     Stack_Type);
--|
--|
-------------------------------------------------------------------
-------------------------------------------------------------------
------------------  P R I V A T E   P A R T   --------------------
-------------------------------------------------------------------
-------------------------------------------------------------------
--|
--|
private

type Node_Type;
--|
--|
--|
type Node_Access_Type is access all Node_Type;
--|
--|
--|
type Node_Type is
   record
   Data          : Data_Type;
   Access_To_Next : Node_Access_Type := null;
end record;
--|
--|  used in pop
--|
procedure Free_Node is new Ada.Unchecked_Deallocation(Object => Node_Type,
                                                      Name   => Node_Access_Type);

--|
--|the stack itself
--|
type Stack_Type
   is
   record
   Node_Access     : Node_Access_Type := null;
   How_Many_Stored : Natural          := 0;
end record;

--|
--|  iterator
```

```
   --|
   type Iterator_Type is
      record
      Current_Node_Access : Node_Access_Type := null;
   end record;

   type Statistics_Type is
      record
      Pop_Calls  : Integer := 0;
      Push_Calls : Integer := 0;
   end record;

   Statistics : Statistics_Type;

end Linked_Stack;---------------------------------------
   --|
   --|

   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |             Initialize              |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   procedure Initialize(Iterator      :    out Iterator_Type;
                        For_The_Stack : in     Stack_Type);
   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |              Is_Done                |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   function Is_Done(Iterator : in     Iterator_Type)
   return Boolean;
   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |      Current_Value                  |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   function Current_Value(Iterator : in     Iterator_Type)
   return Data_Type;
   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |      To_Next_Value                  |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   procedure To_Next_Value(Iterator : in out Iterator_Type);
   --|
   --|
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   ------------------   S T A T I S T I C  S    ---------------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   --|
   --|

   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |       Print_Statistics              |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   procedure Print_Statistics(Stack : in     Stack_Type);
   --|
   --|
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   ------------------  P R I V A T E   P A R T   --------------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   --|
   --|
   private

   type Node_Type;
   --|
   --|
   --|
   type Node_Access_Type is access all Node_Type;
   --|
   --|
   --|
   type Node_Type is
      record
      Data           : Data_Type;
      Access_To_Next : Node_Access_Type := null;
   end record;
   --|
   --|  used in pop
```

```
   --|
   procedure Free_Node is new Ada.Unchecked_Deallocation(Object => Node_Type,
                                                         Name   => Node_Access_Type);

   --|
   --|the stack itself
   --|
   type Stack_Type
      is
      record
      Node_Access     : Node_Access_Type := null;
      How_Many_Stored : Natural          := 0;
   end record;

   --|
   --|  iterator
   --|
   type Iterator_Type is
      record
      Current_Node_Access : Node_Access_Type := null;
   end record;

   type Statistics_Type is
      record
      Pop_Calls  : Integer := 0;
      Push_Calls : Integer := 0;
   end record;

   Statistics : Statistics_Type;

end Linked_Stack;
```

Fichier .adb :

```
package body Linked_Stack
  is
   --|
   --|
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   ------------------   S T A C K   O P E R A T I O N S    ----------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   --|
   --|

   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |                push                 |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   procedure Push(Data         : in     Data_Type;
                  In_The_Stack : in out Stack_Type)
      is
      begin
      In_The_Stack.How_Many_Stored := 1 + In_The_Stack.How_Many_Stored;
      In_The_Stack.Node_Access     := new Node_Type'(Data        => Data,
                                                     Access_To_Next => In_The_Stack.Node_Access);
   end Push;
   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |                pop                  |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   procedure Pop(The_Stack : in out Stack_Type)
      is
      Z : Node_Access_Type := null;
      begin
      Z                          := The_Stack.Node_Access;
      The_Stack.Node_Access      := The_Stack.Node_Access.Access_To_Next;
      The_Stack.How_Many_Stored := The_Stack.How_Many_Stored - 1;

      Free_Node(X => Z);

   end Pop;
   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |             Size_Of                 |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   function Size_Of(The_Stack : in     Stack_Type)
      return Integer
      is
      begin
      return The_Stack.How_Many_Stored;
   end Size_Of;
   --|
   --|                      |_____________________________________|
   --|                      |                                     |
```

```
--|                       |            top_Of              |
--|                       |_________________________________|
--|                       |                                 |
--|
function Top_Of(The_Stack : in     Stack_Type)
   return Data_Type
   is
   begin
   return The_Stack.Node_Access.Data;
end Top_Of;
--|                       |_________________________________|
--|                       |                                 |
--|                       |         Initialize              |
--|                       |_________________________________|
--|                       |                                 |
--|
procedure Initialize(The_Stack : in out Stack_Type)
   is
   Z : Node_Access_Type := The_Stack.Node_Access;
   begin

   while null /= Z loop

      Free_Node(X => Z);
      The_Stack.Node_Access := The_Stack.Node_Access.Access_To_Next;
      Z                     := The_Stack.Node_Access;

   end loop;

   The_Stack.How_Many_Stored := 0;

end Initialize;
--|
--|
-------------------------------------------------------------------
-------------------------------------------------------------------
------------------          I T E R A T O R     ------------------
-------------------------------------------------------------------
-------------------------------------------------------------------
--|
--|

--|
--|                       |_________________________________|
--|                       |                                 |
--|                       |             Initialize          |
--|                       |_________________________________|
--|                       |                                 |
--|
procedure Initialize(Iterator      :    out Iterator_Type;
                     For_The_Stack : in     Stack_Type)
   is
   begin
   Iterator.Current_Node_Access := For_The_Stack.Node_Access;
end Initialize;
--|
--|                       |_________________________________|
--|                       |                                 |
--|                       |            Is_Done              |
--|                       |_________________________________|
--|                       |                                 |
--|
function Is_Done(Iterator : in     Iterator_Type)
   return Boolean
   is
   begin
   return Iterator.Current_Node_Access = null;
end Is_Done;
--|
--|                       |_________________________________|
--|                       |                                 |
--|                       |      Current_Value              |
--|                       |_________________________________|
--|                       |                                 |
--|
function Current_Value(Iterator : in     Iterator_Type)
   return Data_Type
   is
   begin
   return Iterator.Current_Node_Access.Data;
end Current_Value;
--|
--|                       |_________________________________|
--|                       |                                 |
--|                       |      To_Next_Value              |
--|                       |_________________________________|
--|                       |                                 |
--|
procedure To_Next_Value(Iterator : in out Iterator_Type)
   is
   begin
   Iterator.Current_Node_Access := Iterator.Current_Node_Access.Access_To_Next;
end To_Next_Value;
--|
--|
-------------------------------------------------------------------
-------------------------------------------------------------------
```

```
   ------------------   S T A T I S T I C  S   --------------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   --|
   --|

   --|
   --|                     |_____________________________________|
   --|                     |                                     |
   --|                     |       Print_Statistics              |
   --|                     |_____________________________________|
   --|                     |                                     |
   --|
   procedure Print_Statistics(Stack : in     Stack_Type)
      is
      begin
      null;
   end Print_Statistics;
   --|
   --|
   --|
end Linked_Stack;
```

14.6 LE CODE DE LA QUEUE (TABLEAU)

Fichier .ads :

```
  generic

  type Data_Type is private;

  type Index_Type is mod <>;

  package Simple_Queue
    is

    Queue_Full  : exception;
    Queue_Empty : exception;

    type Iterator_Type is limited private;
    type Queue_Type is limited private;

    --|
    --|
    ------------------------------------------------------------------
    ------------------------------------------------------------------
    ------------------   Q U E U E   O P E R A T I O N S   ----------
    ------------------------------------------------------------------
    ------------------------------------------------------------------
    --|
    --|

    --|
    --|                     |_____________________________________|
    --|                     |                                     |
    --|                     |       Add_To_Tail                   |
    --|                     |_____________________________________|
    --|                     |                                     |
    --|
    procedure Add_To_Tail(Data         : in     Data_Type;
                          To_The_Queue : in out Queue_Type);
    --|
    --|                     |_____________________________________|
    --|                     |                                     |
    --|                     |   Remove_From_Head                  |
    --|                     |_____________________________________|
    --|                     |                                     |
    --|
    procedure Remove_From_Head(The_Queue : in out Queue_Type);
    --|
    --|                     |_____________________________________|
    --|                     |                                     |
    --|                     |            Size_Of                  |
    --|                     |_____________________________________|
    --|                     |                                     |
    --|
    function Size_Of(The_Queue : in     Queue_Type)
    return Integer;
    --|
    --|                     |_____________________________________|
    --|                     |                                     |
    --|                     |          Head_Of                    |
    --|                     |_____________________________________|
    --|                     |                                     |
    --|
    function Head_Of(The_Queue : in     Queue_Type)
    return Data_Type;
    --|                     |_____________________________________|
    --|                     |                                     |
    --|                     |         Initialize                  |
```

```
--|                     |_____________________________________|
--|                     |                                     |
--|
procedure Initialize(The_Queue : in out Queue_Type);
--|
--|
-----------------------------------------------------------------
-----------------------------------------------------------------
------------------          I T E R A T O R      ------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
--|
--|

--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |              Initialize             |
--|                     |_____________________________________|
--|                     |                                     |
--|
procedure Initialize(Iterator      :    out Iterator_Type;
                     For_The_Queue : in     Queue_Type);
--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |             Is_Done                 |
--|                     |_____________________________________|
--|                     |                                     |
--|
function Is_Done(Iterator : in     Iterator_Type)
return Boolean;
--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |      Current_Value                  |
--|                     |_____________________________________|
--|                     |                                     |
--|
function Current_Value(Iterator : in     Iterator_Type)
return Data_Type;
--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |      To_Next_Value                  |
--|                     |_____________________________________|
--|                     |                                     |
--|
procedure To_Next_Value(Iterator : in out Iterator_Type);
--|
--|
-----------------------------------------------------------------
-----------------------------------------------------------------
------------------   S T A T I S T I C  S    --------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
--|
--|

--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |       Print_Statistics              |
--|                     |_____________________________________|
--|                     |                                     |
--|
procedure Print_Statistics(Queue : in     Queue_Type);
--|
--|
-----------------------------------------------------------------
-----------------------------------------------------------------
------------------  P R I V A T E   P A R T   --------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
--|
--|
private
--|
--|  queue table
--|
type Queue_Table_Array_Type is array (Index_Type range <>) of Data_Type;
--|
--| declare queue
--|
type Queue_Type
   is
   record
   Queue_Table_Array : aliased Queue_Table_Array_Type(Index_Type'Range);
   Head_Index        : Index_Type := 0;
   Tail_Index        : Index_Type := 0;
end record;
--|
--|  needed for the iterator
--|
type Queue_Table_Array_Access_Type is access all Queue_Table_Array_Type;
--|
--|  iterator
```

```
   --|
   type Iterator_Type is
      record
      Current_Iterator_Index   : Index_Type                   := 0;
      Queue_Iterator_Index     : Index_Type                   := 0;
      Queue_Table_Array_Access : Queue_Table_Array_Access_Type := null;
   end record;


   type Statistics_Type is
      record
      Collisions_Add    : Integer := 0;
      Collisions_Delete : Integer := 0;
      Collisions_Seek   : Integer := 0;
      Add_Calls         : Integer := 0;
      Delete_Calls      : Integer := 0;
      Seek_Calls        : Integer := 0;
   end record;

   Statistics : Statistics_Type;

end Simple_Queue;
```

Fichier .adb :

```
package body Simple_Queue
   is


   --|
   --|
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   ------------------   Q U E U E    O P E R A T I O N S    ----------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   --|
   --|

   --|
   --|                     |_____________________________________|
   --|                     |                                     |
   --|                     |        Add_To_Tail                  |
   --|                     |_____________________________________|
   --|                     |                                     |
   --|
   procedure Add_To_Tail(Data         : in     Data_Type;
                         To_The_Queue : in out Queue_Type)
      is
      begin

      if To_The_Queue.Tail_Index + 1 = To_The_Queue.Head_Index
         then
         raise Queue_Full;
      end if;

      To_The_Queue.Queue_Table_Array(To_The_Queue.Tail_Index) := Data;
      To_The_Queue.Tail_Index                                := 1 + To_The_Queue.Tail_Index;
   end Add_To_Tail;
   --|
   --|                     |_____________________________________|
   --|                     |                                     |
   --|                     |   Remove_From_Head                  |
   --|                     |_____________________________________|
   --|                     |                                     |
   --|
   procedure Remove_From_Head(The_Queue : in out Queue_Type)
      is
      begin

      if The_Queue.Tail_Index = The_Queue.Head_Index
         then
         raise Queue_Empty;
      end if;

      The_Queue.Head_Index := 1 + The_Queue.Head_Index;

   end Remove_From_Head;
   --|
   --|                     |_____________________________________|
   --|                     |                                     |
   --|                     |            Size_Of                  |
   --|                     |_____________________________________|
   --|                     |                                     |
   --|
   function Size_Of(The_Queue : in     Queue_Type)
      return Integer
      is
      Size   : Integer;
      Modulo : Integer := Integer(Index_Type'Last) + 1;
      begin
      Size := Modulo + Integer(The_Queue.Tail_Index) - Integer(The_Queue.Head_Index);
      Size := Size mod Modulo;
      return Size;
```

```
end Size_Of;
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |              Head_Of                |
--|                      |_____________________________________|
--|                      |                                     |
--|
function Head_Of(The_Queue : in     Queue_Type)
   return Data_Type
   is
   begin
   return The_Queue.Queue_Table_Array(The_Queue.Head_Index);
end Head_Of;
--|                      |_____________________________________|
--|                      |                                     |
--|                      |            Initialize               |
--|                      |_____________________________________|
--|                      |                                     |
--|
procedure Initialize(The_Queue : in out Queue_Type)
   is
   begin
   The_Queue.Tail_Index := The_Queue.Head_Index;
end Initialize;
--|
--|
-----------------------------------------------------------------
-----------------------------------------------------------------
------------------          I T E R A T O R     ------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
--|
--|

--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |                Initialize           |
--|                      |_____________________________________|
--|                      |                                     |
--|
procedure Initialize(Iterator      :    out Iterator_Type;
                     For_The_Queue : in     Queue_Type)
   is

   begin

   Iterator := (Current_Iterator_Index   => For_The_Queue.Head_Index,
                Queue_Iterator_Index     => For_The_Queue.Tail_Index,
                Queue_Table_Array_Access => new Queue_Table_Array_Type'(For_The_Queue.Queue_Table_Array));

end Initialize;
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |              Is_Done                |
--|                      |_____________________________________|
--|                      |                                     |
--|
function Is_Done(Iterator : in     Iterator_Type)
   return Boolean
   is
   begin
   return Iterator.Current_Iterator_Index = Iterator.Queue_Iterator_Index;
end Is_Done;
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |       Current_Value                 |
--|                      |_____________________________________|
--|                      |                                     |
--|
function Current_Value(Iterator : in     Iterator_Type)
   return Data_Type
   is
   begin
   return Iterator.Queue_Table_Array_Access.all (Iterator.Current_Iterator_Index);
end Current_Value;
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |       To_Next_Value                 |
--|                      |_____________________________________|
--|                      |                                     |
--|
procedure To_Next_Value(Iterator : in out Iterator_Type)
   is
   begin
   Iterator.Current_Iterator_Index := 1 + Iterator.Current_Iterator_Index;
end To_Next_Value;
--|
--|
-----------------------------------------------------------------
-----------------------------------------------------------------
------------------   S T A T I S T I C  S    --------------------
-----------------------------------------------------------------
```

```
   ------------------------------------------------------------------
   --|
   --|

   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |       Print_Statistics              |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   procedure Print_Statistics(Queue : in     Queue_Type)
      is
      begin
      null;
   end Print_Statistics;


end Simple_Queue;
```

14.7 LE CODE DE LA QUEUE (POINTEURS)

Fichier linked_queue.ads :

```
with Ada.Unchecked_Deallocation;

generic

type Data_Type is private;


package Linked_Queue
  is


  Queue_Empty : exception;

  type Iterator_Type is limited private;
  type Queue_Type is limited private;

  --|
  --|
  ------------------------------------------------------------------
  ------------------------------------------------------------------
  ------------------   Q U E U E    O P E R A T I O N S    ----------
  ------------------------------------------------------------------
  ------------------------------------------------------------------
  --|
  --|

  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |        Add_To_Tail                  |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  procedure Add_To_Tail(Data         : in     Data_Type;
                        To_The_Queue : in out Queue_Type);
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |   Remove_From_Head                  |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  procedure Remove_From_Head(The_Queue : in out Queue_Type);
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |            Size_Of                  |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  function Size_Of(The_Queue : in     Queue_Type)
  return Integer;
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |           Head_Of                   |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  function Head_Of(The_Queue : in     Queue_Type)
  return Data_Type;
  --|                      |_____________________________________|
  --|                      |                                     |
```

```
--|                     |         Initialize             |
--|                     |________________________________|
--|                     |                                |
--|
procedure Initialize(The_Queue : in out Queue_Type);
--|
--|
-------------------------------------------------------------------
-------------------------------------------------------------------
------------------          I T E R A T O R     ------------------
-------------------------------------------------------------------
-------------------------------------------------------------------
--|
--|

--|
--|                     |________________________________|
--|                     |                                |
--|                     |             Initialize         |
--|                     |________________________________|
--|                     |                                |
--|
procedure Initialize(Iterator      :    out Iterator_Type;
                     For_The_Queue : in     Queue_Type);
--|
--|                     |________________________________|
--|                     |                                |
--|                     |            Is_Done             |
--|                     |________________________________|
--|                     |                                |
--|
function Is_Done(Iterator : in     Iterator_Type)
return Boolean;
--|
--|                     |________________________________|
--|                     |                                |
--|                     |      Current_Value             |
--|                     |________________________________|
--|                     |                                |
--|
function Current_Value(Iterator : in     Iterator_Type)
return Data_Type;
--|
--|                     |________________________________|
--|                     |                                |
--|                     |      To_Next_Value             |
--|                     |________________________________|
--|                     |                                |
--|
procedure To_Next_Value(Iterator : in out Iterator_Type);
--|
--|
-------------------------------------------------------------------
-------------------------------------------------------------------
------------------   S T A T I S T I C  S    ---------------------
-------------------------------------------------------------------
-------------------------------------------------------------------
--|
--|

--|
--|                     |________________________________|
--|                     |                                |
--|                     |       Print_Statistics         |
--|                     |________________________________|
--|                     |                                |
--|
procedure Print_Statistics(Queue : in     Queue_Type);
--|
--|
-------------------------------------------------------------------
-------------------------------------------------------------------
------------------  P R I V A T E   P A R T   --------------------
-------------------------------------------------------------------
-------------------------------------------------------------------
--|
--|
private

type Node_Type;
--|
--|
--|
type Node_Access_Type is access all Node_Type;
--|
--|
--|
type Node_Type is
   record
   Data           : Data_Type;
   Access_To_Next : Node_Access_Type := null;
end record;
--|
--|  used in deletions
--|
procedure Free_Node is new Ada.Unchecked_Deallocation(Object => Node_Type,
                                                      Name   => Node_Access_Type);
```

```
   --|
   --|the queue itself
   --|
   type Queue_Type
      is
      record
      Head_Node_Access : Node_Access_Type := null;
      Tail_Node_Access : Node_Access_Type := null;
      How_Many_Stored  : Natural          := 0;
   end record;

   --|
   --|  iterator
   --|
   type Iterator_Type is
      record
      Current_Node_Access : Node_Access_Type := null;
   end record;

   type Statistics_Type is
      record
      Add_To_Tail_Calls      : Integer := 0;
      Remove_From_Head_Calls : Integer := 0;
   end record;

   Statistics : Statistics_Type;

end Linked_Queue;
```

Fichier linked_queue.adb :

```
package body Linked_Queue
   is

   --|
   --|
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   ------------------   Q U E U E    O P E R A T I O N S    ----------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   --|
   --|

   --|
   --|                     |_____________________________________|
   --|                     |                                     |
   --|                     |          Add_To_Tail                |
   --|                     |_____________________________________|
   --|                     |                                     |
   --|
   procedure Add_To_Tail(Data         : in     Data_Type;
                         To_The_Queue : in out Queue_Type)
      is
      begin
      --
      --========== starts empty ==========
      --
      -- H -----> null
      -- T -----> null
      --
      --========== added first one  ==========
      --
      -- H ----->| first |----> null
      --          ^
      -- T --------|
      --
      ----========== added second one  ==========
      --
      -- H ----->| first |----
      --                     |
      --            ---------|
      --            |
      --            V
      -- T ------>|  2nd  |----> null
      --
      --
      ----========== added third one  ==========

      --
      -- H ----->| first |----
      --                     |
      --            ---------|
      --            |
      --            V
      --         |  2nd  |----
      --                     |
      --            ---------|
      --            |
      --            V
      -- T ------>|  3rd  |----> null
      --
      --
      --
```

```
      --  and so one and so forth
      --
      --


      if 0 = To_The_Queue.How_Many_Stored
         then
         --
         -- ======= from
         --
         -- H -----> null
         -- T -----> null
         --
         -- ======= to
         --
         -- H ----->| first |----> null
         --           ^
         -- T --------|
         --
         To_The_Queue.Head_Node_Access := new Node_Type'(Data           => Data,
                                                         Access_To_Next => null);
         To_The_Queue.Tail_Node_Access := To_The_Queue.Head_Node_Access;

         else
         --
         --========== starting at  ==========
         --
         -- H ----->| first |----
         --                   |
         --             ---------|
         --             |
         --             V
         -- T ----->|  2nd  |----> null
         --
         --========== after 1 st step  ==========
         --
         --
         -- H ----->| first |----
         --                   |
         --             ---------|
         --             |
         --             V
         -- T ----->|  2nd  |---
         --                   |
         --             ---------|
         --             |
         --             V
         --           |  3rd  |----> null
         --
         To_The_Queue.Tail_Node_Access.Access_To_Next := new Node_Type'(Data           => Data,
                                                                        Access_To_Next => null);

         --
         --========== after 2 nd step  ==========
         --
         -- H ----->| first |----
         --                    |
         --             ---------|
         --             |
         --             V
         --          |  2nd  |----
         --                    |
         --             ---------|
         --             |
         --             V
         -- T ----->|  3rd  |----> null
         --
         --
         To_The_Queue.Tail_Node_Access := To_The_Queue.Tail_Node_Access.Access_To_Next;
      end if;

      To_The_Queue.How_Many_Stored := 1 + To_The_Queue.How_Many_Stored;

   end Add_To_Tail;
   --|
   --|                     |_____________________________________|
   --|                     |                                     |
   --|                     |   Remove_From_Head                  |
   --|                     |_____________________________________|
   --|                     |                                     |
   --|
   procedure Remove_From_Head(The_Queue : in out Queue_Type)
      is
      Z : Node_Access_Type := null;
      begin

      case The_Queue.How_Many_Stored
         is
         when 0 => raise Queue_Empty;

         when 1 =>
         --
         -- ======= from
         --
         -- H ----->| first |----> null
         --           ^
         -- T --------|
```

```
      --
      --
      -- ======= to
      --
      -- H -----> null
      -- T -----> null
      --
      Z                          := The_Queue.Head_Node_Access;
      The_Queue.Head_Node_Access := null;
      The_Queue.Tail_Node_Access := null;

      Free_Node(X => Z);


      when others =>
      --
      --============ from
      -- H ----->| first |----
      --                  |
      --            ---------|
      --            |
      --            V
      --         |  2nd  |----
      --                  |
      --            ---------|
      --            |
      --            V
      -- T ----->|  3rd  |----> null
      --
      --
      --
      --========== to  ==========
      --
      -- H ----->|  2nd  |----
      --                  |
      --            ---------|
      --            |
      --            V
      -- T ----->|  3rd  |----> null
      --
      --
      Z                          := The_Queue.Head_Node_Access;
      The_Queue.Head_Node_Access := The_Queue.Head_Node_Access.Access_To_Next;
      Free_Node(X => Z);
   end case;

   The_Queue.How_Many_Stored := The_Queue.How_Many_Stored - 1;

end Remove_From_Head;
--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |            Size_Of                  |
--|                     |_____________________________________|
--|                     |                                     |
--|
function Size_Of(The_Queue : in     Queue_Type)
   return Integer
   is
   begin
   return The_Queue.How_Many_Stored;
end Size_Of;
--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |           Head_Of                   |
--|                     |_____________________________________|
--|                     |                                     |
--|
function Head_Of(The_Queue : in     Queue_Type)
   return Data_Type
   is
   begin
   return The_Queue.Head_Node_Access.Data;
end Head_Of;
--|                     |_____________________________________|
--|                     |                                     |
--|                     |         Initialize                  |
--|                     |_____________________________________|
--|                     |                                     |
--|
procedure Initialize(The_Queue : in out Queue_Type)
   is
   begin
   while The_Queue.How_Many_Stored /= 0 loop
      Remove_From_Head(The_Queue => The_Queue);
   end loop;
end Initialize;
--|
--|
-------------------------------------------------------------------
-------------------------------------------------------------------
------------------          I T E R A T O R     ------------------
-------------------------------------------------------------------
-------------------------------------------------------------------
--|
--|
```

```
   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |             Initialize              |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   procedure Initialize(Iterator      :    out Iterator_Type;
                        For_The_Queue : in     Queue_Type)
      is
      begin
      Iterator.Current_Node_Access := For_The_Queue.Head_Node_Access;
   end Initialize;
   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |              Is_Done                |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   function Is_Done(Iterator : in     Iterator_Type)
      return Boolean
      is
      begin
      return Iterator.Current_Node_Access = null;
   end Is_Done;
   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |       Current_Value                 |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   function Current_Value(Iterator : in     Iterator_Type)
      return Data_Type
      is
      begin
      return Iterator.Current_Node_Access.Data;
   end Current_Value;
   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |       To_Next_Value                 |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   procedure To_Next_Value(Iterator : in out Iterator_Type)
      is
      begin
      Iterator.Current_Node_Access := Iterator.Current_Node_Access.Access_To_Next;
   end To_Next_Value;
   --|
   --|
   ------------------------------------------------------------------------
   ------------------------------------------------------------------------
   -------------------   S T A T I S T I C  S    --------------------------
   ------------------------------------------------------------------------
   ------------------------------------------------------------------------
   --|
   --|

   --|
   --|                      |_____________________________________|
   --|                      |                                     |
   --|                      |        Print_Statistics             |
   --|                      |_____________________________________|
   --|                      |                                     |
   --|
   procedure Print_Statistics(Queue : in     Queue_Type)
      is
      begin
      null;
   end Print_Statistics;
   --|
   --|
 end Linked_Queue;
```

14.8 LE CODE DE L'ANNEAU

Fichier linked_ring.ads :

```
with Ada.Unchecked_Deallocation;

generic

type Data_Type is private;
```

```
package Linked_Ring
  is


  Ring_Empty     : exception;
  Bookmark_Empty : exception;

  type Iterator_Type is limited private;
  type Ring_Type is limited private;

  type Direction_Type is (Left_Sided, Right_Sided);

  --|
  --|              |-------|
  --|              | Clear |
  --|              |_______|
  --|
  procedure Clear(The_Ring : in out Ring_Type);
  --|
  --|              |-------------------------------|
  --|              |           Insert              |
  --|              |_______________________________|
  --|
  procedure Insert(The_Item    : in     Data_Type;
                   In_The_Ring : in out Ring_Type;
                   Direction   : in     Direction_Type := Left_Sided);
  --|
  --|              |-------------------------------|
  --|              |            Delete_Top         |
  --|              |_______________________________|
  --|
  procedure Delete_Top(In_The_Ring : in out Ring_Type;
                       Direction   : in     Direction_Type := Left_Sided);
  --|
  --|              |-------------------------------|
  --|              |            Top_Of             |
  --|              |_______________________________|
  --|
  function Top_Of(The_Ring : in     Ring_Type) return Data_Type;
  --|
  --|              |-------------------------------|
  --|              |         Size_Of               |
  --|              |_______________________________|
  --|
  function Size_Of(The_Ring : in     Ring_Type) return Integer;
  --|
  --|              |-------------------------------|
  --|              |            rotate             |
  --|              |_______________________________|
  --|
  procedure Rotate(The_Ring  : in out Ring_Type;
                   Direction : in     Direction_Type := Left_Sided);
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |           Set_Bookmark              |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  procedure Set_Bookmark(For_The_Ring : in out Ring_Type);
  --|
  --|              |-------------------------------|
  --|              |    Bookmark_Value             |
  --|              |_______________________________|
  --|
  function Bookmark_Value(The_Ring : in     Ring_Type) return Data_Type;

  ------------------------------------------------------------------------
  ------------------------------------------------------------------------
  ------------------          I T E R A T O R     ------------------
  ------------------------------------------------------------------------
  ------------------------------------------------------------------------
  --|
  --|

  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |             Initialize              |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  procedure Initialize(Iterator     :    out Iterator_Type;
                       For_The_Ring : in out Ring_Type;
                       Direction    : in     Direction_Type := Left_Sided);
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |             Is_Done                 |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  function Is_Done(Iterator : in     Iterator_Type) return Boolean;
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |       Current_Value                 |
```

```
   --|                     |_____________________________|
   --|                     |                             |
   --|
   function Current_Value(Iterator : in     Iterator_Type) return Data_Type;
   --|
   --|                     |_____________________________|
   --|                     |                             |
   --|                     |      To_Next_Value          |
   --|                     |_____________________________|
   --|                     |                             |
   --|
   procedure To_Next_Value(Iterator : in out Iterator_Type);
   --|
   --|
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   -------------------   S T A T I S T I C  S    --------------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   --|
   --|

   --|
   --|                     |_____________________________|
   --|                     |                             |
   --|                     |       Print_Statistics      |
   --|                     |_____________________________|
   --|                     |                             |
   --|
   procedure Print_Statistics(Ring : in     Ring_Type);
   --|
   --|
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   -------------------       P R I V A T E       --------------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   --|
   --|
   private
   --|
   --|
   --|
   type Node_Type;
   --|
   --|
   --|
   type Node_Access_Type is access all Node_Type;
   --|
   --|
   --|
   type Ring_Type is
      record
      Access_To_Top      : Node_Access_Type := null;
      Access_To_Bookmark : Node_Access_Type := null;
      How_Many_Stored    : Integer          := 0;
   end record;
   --|
   --|
   --|
   type Node_Type is
      record
      Data  : Data_Type;
      Left  : Node_Access_Type := null;
      Right : Node_Access_Type := null;
   end record;

   type Iterator_Type is
      record
      Current_Node_Access : Node_Access_Type;
      Count               : Integer        := 0;
      Count_Until         : Integer        := 0;
      Side                : Direction_Type := Direction_Type'First;
   end record;
   --|
   --|  used in deletions
   --|
   procedure Free_Node is new Ada.Unchecked_Deallocation(Object => Node_Type,
                                                        Name   => Node_Access_Type);

end Linked_Ring;
```

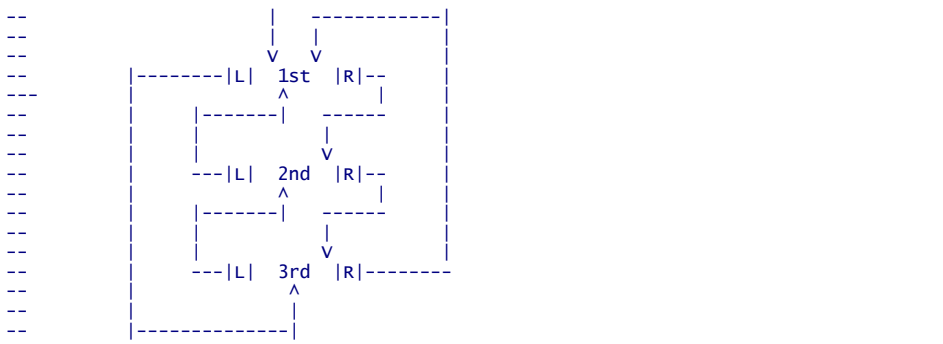
Fichier linked_ring.adb :

```
package body Linked_Ring
  is
   --|
   --|
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   -------------------   R I N G     O P E R A T I O N S    ----------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
```

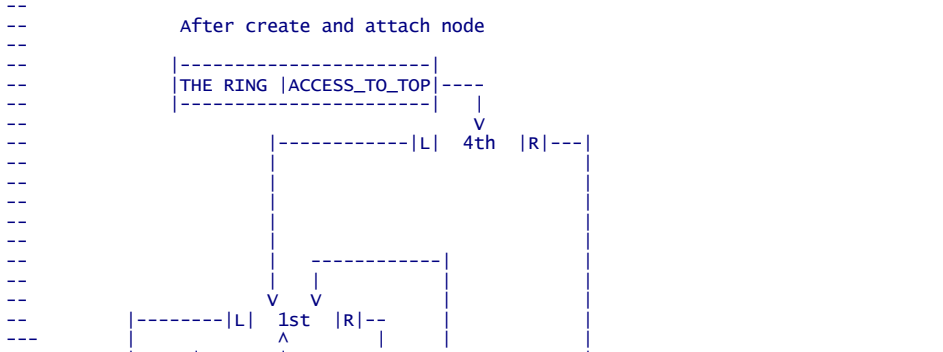
```
--|
--|

--|              |-------|
--|              | Clear |
--|              |_______|
--|
procedure Clear(The_Ring : in out Ring_Type)
   is
   begin

   while not (0 = The_Ring.How_Many_Stored) loop

      Delete_Top(In_The_Ring => The_Ring);

   end loop;

end Clear;
--|
--|              |-------------------------------|
--|              | Insert_At_The_Left_Of_The_Top |
--|              |_______________________________|
--|

--
--
--
--             |----------------------|
--             |THE RING |ACCESS_TO_TOP|---|
--             |----------------------|   |
--                                        |
--                      |------------------|
--                      |
--                      |   ------------|
--                      |   |           |
--                     V   V           |
--          |--------|L|  1st  |R|--     |
---         |           ^       |       |
--          |     |-------|   ------     |
--          |     |           |          |
--          |     |           V          |
--          |     ---|L|  2nd  |R|--     |
--          |           ^       |       |
--          |     |-------|   ------     |
--          |     |           |          |
--          |     |           V          |
--          |     ---|L|  3rd  |R|--------
--          |              ^
--          |              |
--          |--------------|
--

procedure Insert_At_The_Left_Of_The_Top(The_Item    : in     Data_Type;
                                        In_The_Ring : in out Ring_Type)
   is
   begin
   --|
   --|  first time ?
   --|
   if In_The_Ring.Access_To_Top = null
      then
      --|
      --|create and attach node
      --|
      In_The_Ring.Access_To_Top := new Node_Type'(Left  => null,
                                                  Data  => The_Item,
                                                  Right => null);
      --|
      --| loop on itself
      --|
      In_The_Ring.Access_To_Top.Left  := In_The_Ring.Access_To_Top;
      In_The_Ring.Access_To_Top.Right := In_The_Ring.Access_To_Top;

      --|
      --|  Nth  time ?
      --|

      --                  S T A R T I N G   A T
      --
      --
      --             |----------------------|
      --             |THE RING |ACCESS_TO_TOP|---|
      --             |----------------------|   |
      --                                        |
      --                      |------------------|
      --                      |
      --                      |   ------------|
      --                      |   |           |
      --                     V   V           |
      --          |--------|L|  1st  |R|--     |
      ---         |           ^       |       |
      --          |     |-------|   ------     |
      --          |     |           |          |
      --          |     |           V          |
      --          |     ---|L|  2nd  |R|--     |
      --          |           ^       |       |
      --          |     |-------|   ------     |
```

```
--         |     |          |         |
--         |     |          V         |
--         |     ---|L|  3rd  |R|--------
--         |             ^
--         |             |
--         |-------------|
--
else
--|
--|create and attach node
--|
In_The_Ring.Access_To_Top := new Node_Type'(Left  => In_The_Ring.Access_To_Top.Left,
                                            Data  => The_Item,
                                            Right => In_The_Ring.Access_To_Top);
--
--              After create and attach node
--
--            |-----------------------|
--            |THE RING |ACCESS_TO_TOP|----
--            |-----------------------|   |
--                                       V
--         |-------------------------|L|  4th  |R|---|
--         |                                         |
--         |                                         |
--         |                                         |
--         |            |-----------------------------|
--         |            |
--         |            |   ------------|
--         |            |   |           |
--         |           V   V           |
--         |--------|L|  1st  |R|--     |
---        |            ^        |      |
--         |     |-------|   ------      |
--         |     |           |           |
--         |     |           V           |
--         |     ---|L|  2nd  |R|--      |
--         |            ^        |      |
--         |     |-------|   ------      |
--         |     |           |           |
--         |     |           V           |
--         |     ---|L|  3rd  |R|--------
--         |            ^
--         |            |
--         |-------------|
--

--|
--| insert
--|
In_The_Ring.Access_To_Top.Right.Left := In_The_Ring.Access_To_Top;

--
--              After In_The_Ring.Access_To_Top.Right.Left := In_The_Ring.Access_To_Top
--
--            |-----------------------|
--            |THE RING |ACCESS_TO_TOP|----
--            |-----------------------|   |
--                                       V
--         |-------------------------|L|  4th  |R|---|
--         |                                 ^       |
--         |   |-----------------------------|       |
--         |   |                                     |
--         |   |        |-----------------------------|
--         |   |        |
--         |   |        |   ------------|
--         |   |        |   |           |
--         |   |       V   V           |
--         |   -----|L|  1st  |R|--     |
---        |            ^        |      |
--         |     |-------|   ------      |
--         |     |           |           |
--         |     |           V           |
--         |     ---|L|  2nd  |R|--      |
--         |            ^        |      |
--         |     |-------|   ------      |
--         |     |           |           |
--         |     |           V           |
--         |     ---|L|  3rd  |R|--------
--         |            ^
--         |            |
--         |-------------|
--

In_The_Ring.Access_To_Top.Left.Right := In_The_Ring.Access_To_Top;

--
--              After In_The_Ring.Access_To_Top.Right.Left := In_The_Ring.Access_To_Tope
--
--            |-----------------------|
--            |THE RING |ACCESS_TO_TOP|----   |---------------|
--            |-----------------------|   |   |               |
--                                       V   V               |
--         |-------------------------|L|  4th  |R|---|       |
--         |                                 ^       |       |
--         |   |-----------------------------|       |       |
--         |   |                                     |       |
--         |   |        |-----------------------------|       |
--
```

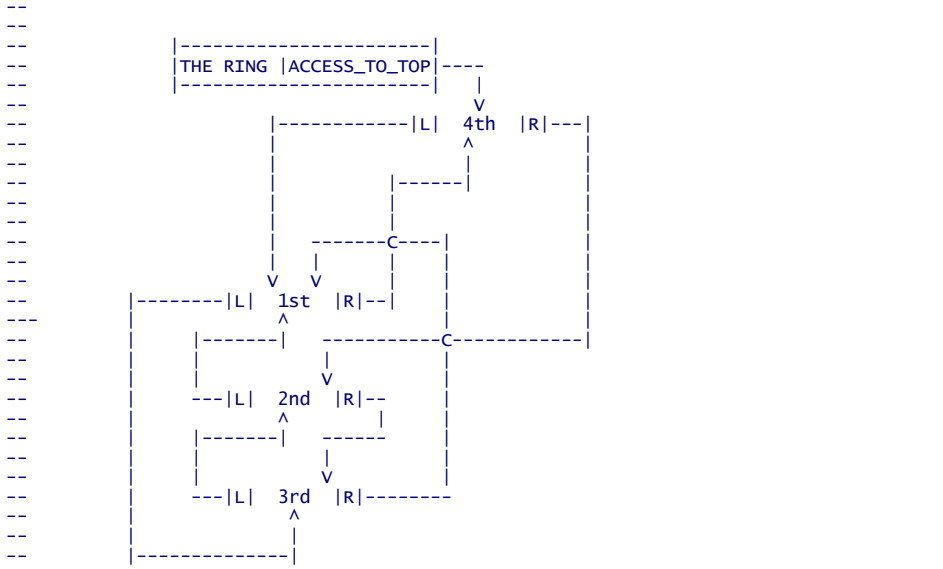
```
      --         |   |        |                                      |
      --         |   |        |                                      |
      --         |   |        |                                      |
      --         |   |       V                                       |
      --         |   -----|L|  1st  |R|--                           |
      ---        |          ^       |                                |
      --         |     |-------|   ------                            |
      --         |     |           |                                 |
      --         |     |          V                                  |
      --         |     ---|L|  2nd  |R|--                           |
      --         |          ^       |                                |
      --         |     |-------|   ------                            |
      --         |     |           |                                 |
      --         |     |          V                                  |
      --         |     ---|L|  3rd  |R|-------------------------------
      --         |          ^
      --         |          |
      --         |--------------|
      --

      --                  ENDING WITH  (same as above redrawn)
      --
      --
      --            |-----------------------|
      --            |THE RING |ACCESS_TO_TOP|---|
      --            |-----------------------|   |
      --                                        |
      --                      |-----------------|
      --                      |
      --                      |   ------------|
      --                      |   |           |
      --                     V   V           |
      --         |--------|L|  4th  |R|--     |
      ---        |          ^       |        |
      --         |     |-------|   ------     |
      --         |     |           |          |
      --         |     |          V          |
      --         |     ---|L|  1st  |R|--     |
      --         |          ^       |        |
      --         |     |-------|   ------     |
      --         |     |           |          |
      --         |     |          V          |
      --         |     ---|L|  2nd  |R|--     |
      --         |          ^       |        |
      --         |     |-------|   ------     |
      --         |     |           |          |
      --         |     |          V          |
      --         |     ---|L|  3rd  |R|--------
      --         |          ^
      --         |          |
      --         |--------------|
      --
   end if;

   In_The_Ring.How_Many_Stored := 1 + In_The_Ring.How_Many_Stored;

end Insert_At_The_Left_Of_The_Top;
--|
--|              |-------------------------------|
--|              | Insert_At_The_Left_Of_The_Top |
--|              |_______________________________|
--|
procedure Insert_At_The_Right_Of_The_Top(The_Item    : in     Data_Type;
                                         In_The_Ring : in out Ring_Type)
   is
   begin
   --|
   --|  first time ?
   --|
   if In_The_Ring.Access_To_Top = null
      then
      --|
      --|create and attach node
      --|
      In_The_Ring.Access_To_Top := new Node_Type'(Right => null,
                                                  Data  => The_Item,
                                                  Left  => null);
      --|
      --| loop on itself
      --|
      In_The_Ring.Access_To_Top.Right := In_The_Ring.Access_To_Top;
      In_The_Ring.Access_To_Top.Left  := In_The_Ring.Access_To_Top;

      --|
      --|  Nth  time ?
      --|

      --                  S T A R T I N G   A T
      --
      --
      --            |-----------------------|
      --            |THE RING |ACCESS_TO_TOP|---|
      --            |-----------------------|   |
      --                                        |
      --                      |-----------------|
      --                      |
```



```
else
--|
--|create and attach node
--|
In_The_Ring.Access_To_Top := new Node_Type'(Left  => In_The_Ring.Access_To_Top,
                                            Data  => The_Item,
                                            Right => In_The_Ring.Access_To_Top.Right);
```



```
--|
--| insert
--|
In_The_Ring.Access_To_Top.Left.Right := In_The_Ring.Access_To_Top;
```



```
In_The_Ring.Access_To_Top.Right.Left := In_The_Ring.Access_To_Top;
```

```
      --
      --              After In_The_Ring.Access_To_Top.LEFT.RIGHT := In_The_Ring.Access_To_Top
      --
      --
      --
      --             |-----------------------|
      --             |THE RING |ACCESS_TO_TOP|----
      --             |-----------------------|    |
      --                                          V
      --                      |------------|L|  4th  |R|---|
      --                      |                 ^         |
      --                      |                 |         |
      --                      |          |------|         |
      --                      |          |                |
      --                      |          |                |
      --                      |   -------C----|           |
      --                      |   |      |    |           |
      --                     V   V      |    |           |
      --          |--------|L|  1st  |R|--|    |           |
      ---         |                     |    |           |
      ---         |                     |    |           |
      ---         |             |---------|    |           |
      ---         |             |              |           |
      --          |     |-------|   -----------C-----------|
      --          |     |           |          |
      --          |     |          V           |
      --          |     ---|L|  2nd  |R|--     |
      --          |              ^       |     |
      --          |     |-------|   ------     |
      --          |     |           |          |
      --          |     |          V           |
      --          |     ---|L|  3rd  |R|--------
      --          |              ^
      --          |              |
      --          |--------------|
      --


      --                   ENDING WITH  (same as above redrawn)
      --
      --
      --             |-----------------------|
      --             |THE RING |ACCESS_TO_TOP|---|
      --             |-----------------------|   |
      --                                         |
      --                      |------------------|
      --                      |
      --                      |   ------------|
      --                      |   |           |
      --                     V   V           |
      --          |--------|L|  4th  |R|--    |
      ---         |              ^       |    |
      --          |     |-------|   ------    |
      --          |     |           |         |
      --          |     |          V          |
      --          |     ---|L|  2nd  |R|--    |
      --          |              ^       |    |
      --          |     |-------|   ------    |
      --          |     |           |         |
      --          |     |          V          |
      --          |     ---|L|  3nd  |R|--    |
      --          |              ^       |    |
      --          |     |-------|   ------    |
      --          |     |           |         |
      --          |     |          V          |
      --          |     ---|L|  1st  |R|--------
      --          |              ^
      --          |              |
      --          |--------------|
      --
   end if;

   In_The_Ring.How_Many_Stored := 1 + In_The_Ring.How_Many_Stored;

end Insert_At_The_Right_Of_The_Top;
--|
--|             |-------------------------------|
--|             |            Insert             |
--|             |_______________________________|
--|
procedure Insert(The_Item    : in     Data_Type;
                 In_The_Ring : in out Ring_Type;
                 Direction   : in     Direction_Type := Left_Sided)
   is
   begin

   if Direction = Left_Sided
      then
      Insert_At_The_Left_Of_The_Top(The_Item    => The_Item,
                                    In_The_Ring => In_The_Ring);
      elsif Direction = Right_Sided
      then

      Insert_At_The_Right_Of_The_Top(The_Item    => The_Item,
                                     In_The_Ring => In_The_Ring);
   end if;
```

```
end Insert;
--|
--|              |--------------|
--|              | Pop_To_Right |
--|              |______________|
--|
procedure Pop_To_Right(The_Ring : in out Ring_Type)
   is
   Z : Node_Access_Type := The_Ring.Access_To_Top;
   begin
   --|
   --|
   --|
   if The_Ring.Access_To_Top = The_Ring.Access_To_Top.Right
      then
      --|
      --|
      --|
      Free_Node(X => The_Ring.Access_To_Top);
      Free_Node(X => The_Ring.Access_To_Bookmark);
      else
      --|
      --|
      --|
      The_Ring.Access_To_Top.Left.Right := The_Ring.Access_To_Top.Right;
      The_Ring.Access_To_Top.Right.Left := The_Ring.Access_To_Top.Left;
      --|
      --|
      --|
      if The_Ring.Access_To_Bookmark = The_Ring.Access_To_Top
         then
         The_Ring.Access_To_Bookmark := null;
      end if;
      --|
      --|
      --|
      The_Ring.Access_To_Top := The_Ring.Access_To_Top.Right;
      Free_Node(X => Z);
   end if;


end Pop_To_Right;
--|
--|              |--------------|
--|              | Pop_To_Left  |
--|              |______________|
--|
procedure Pop_To_Left(The_Ring : in out Ring_Type)
   is
   Z : Node_Access_Type := The_Ring.Access_To_Top;
   begin
   --|
   --|
   --|
   if The_Ring.Access_To_Top = The_Ring.Access_To_Top.Left
      then
      --|
      --|
      --|
      Free_Node(X => The_Ring.Access_To_Top);
      Free_Node(X => The_Ring.Access_To_Bookmark);
      else
      --|
      --|
      --|
      The_Ring.Access_To_Top.Right.Left := The_Ring.Access_To_Top.Left;
      The_Ring.Access_To_Top.Left.Right := The_Ring.Access_To_Top.Right;
      --|
      --|
      --|
      if The_Ring.Access_To_Bookmark = The_Ring.Access_To_Top
         then
         The_Ring.Access_To_Bookmark := null;
      end if;
      --|
      --|
      --|
      The_Ring.Access_To_Top := The_Ring.Access_To_Top.Left;
      Free_Node(X => Z);
   end if;

end Pop_To_Left;
--|
--|              |--------------------------------|
--|              |            Delete_Top          |
--|              |________________________________|
--|
procedure Delete_Top(In_The_Ring : in out Ring_Type;
                     Direction   : in     Direction_Type := Left_Sided)
   is
   begin

   if Direction = Left_Sided
      then
      Pop_To_Left(The_Ring => In_The_Ring);
```

```
      elsif Direction = Right_Sided
      then
      Pop_To_Right(The_Ring => In_The_Ring);
   end if;

   In_The_Ring.How_Many_Stored := In_The_Ring.How_Many_Stored - 1;
end Delete_Top;
--|
--|              |-------------------------------|
--|              |            Top_Of             |
--|              |_______________________________|
--|
function Top_Of(The_Ring : in     Ring_Type)
   return Data_Type
   is
   begin

   if null = The_Ring.Access_To_Top
      then
      raise Ring_Empty;
      else
      return The_Ring.Access_To_Top.all.Data;
   end if;

end Top_Of;
--|
--|              |-------------------------------|
--|              |        Size_Of                |
--|              |_______________________________|
--|
function Size_Of(The_Ring : in     Ring_Type)
   return Integer
   is
   begin
   return The_Ring.How_Many_Stored;
end Size_Of;
--|
--|              |-------------------------------|
--|              |            rotate             |
--|              |_______________________________|
--|
procedure Rotate(The_Ring  : in out Ring_Type;
                 Direction : in     Direction_Type := Left_Sided)
   is
   begin

   if Direction = Left_Sided
      then
      The_Ring.Access_To_Top := The_Ring.Access_To_Top.Left;
      elsif Direction = Right_Sided
      then
      The_Ring.Access_To_Top := The_Ring.Access_To_Top.Right;
   end if;
end Rotate;
--|
--|                      |_____________________________|
--|                      |                             |
--|                      |          Set_Bookmark       |
--|                      |_____________________________|
--|                      |                             |
--|
procedure Set_Bookmark(For_The_Ring : in out Ring_Type)
   is
   begin
   For_The_Ring.Access_To_Bookmark := For_The_Ring.Access_To_Top;

end Set_Bookmark;
--|
--|              |-------------------------------|
--|              |    Bookmark_Value             |
--|              |_______________________________|
--|
function Bookmark_Value(The_Ring : in     Ring_Type)
   return Data_Type
   is
   begin

   if null = The_Ring.Access_To_Bookmark
      then
      raise Bookmark_Empty;
      else
      return The_Ring.Access_To_Bookmark.all.Data;
   end if;

end Bookmark_Value;
--|
--|
-------------------------------------------------------------------
-------------------------------------------------------------------
------------------          I T E R A T O R     ------------------
-------------------------------------------------------------------
-------------------------------------------------------------------
--|
--|

--|
--|                      |_____________________________|
```

```
   --|                     |                           |
   --|                     |            Initialize     |
   --|                     |___________________________|
   --|                     |                           |
   --|
   procedure Initialize(Iterator     :    out Iterator_Type;
                        For_The_Ring : in out Ring_Type;
                        Direction    : in     Direction_Type := Left_Sided)
      is
      begin
      Iterator := (Current_Node_Access => For_The_Ring.Access_To_Top,
                   Count               => 0,
                   Count_Until         => For_The_Ring.How_Many_Stored,
                   Side                => Direction);

      To_Next_Value(Iterator => Iterator);
   end Initialize;
   --|
   --|                     |___________________________|
   --|                     |                           |
   --|                     |           Is_Done         |
   --|                     |___________________________|
   --|                     |                           |
   --|
   function Is_Done(Iterator : in     Iterator_Type)
      return Boolean
      is
      begin
      return Iterator.Count = 1 + Iterator.Count_Until;
   end Is_Done;
   --|
   --|                     |___________________________|
   --|                     |                           |
   --|                     |      Current_Value        |
   --|                     |___________________________|
   --|                     |                           |
   --|
   function Current_Value(Iterator : in     Iterator_Type)
      return Data_Type
      is
      begin
      return Iterator.Current_Node_Access.Data;
   end Current_Value;
   --|
   --|                     |___________________________|
   --|                     |                           |
   --|                     |      To_Next_Value        |
   --|                     |___________________________|
   --|                     |                           |
   --|
   procedure To_Next_Value(Iterator : in out Iterator_Type)
      is
      begin
      if Iterator.Side = Left_Sided
         then
         Iterator.Current_Node_Access := Iterator.Current_Node_Access.Left;
         else
         Iterator.Current_Node_Access := Iterator.Current_Node_Access.Right;
      end if;
      Iterator.Count := 1 + Iterator.Count;
   end To_Next_Value;
   --|
   --|
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   ------------------   S T A T I S T I C  S    ---------------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   --|
   --|

   --|
   --|                     |___________________________|
   --|                     |                           |
   --|                     |       Print_Statistics    |
   --|                     |___________________________|
   --|                     |                           |
   --|
   procedure Print_Statistics(Ring : in     Ring_Type)
      is
      begin
      null;
   end Print_Statistics;
   --|
   --|


end Linked_Ring;
```

14.9 LE CODE DE L'ARBRE BINAIRE (TABLEAU)

Fichier simple_bin_tree.ads :

```
generic

type Data_Type is private;
Max_Number_Of_Items : Positive;
Null_Data           : Data_Type;

--|
--| for = operator
--|
with function "=" (Left  : in Data_Type;
                   Right : in     Data_Type) return Boolean is <>;
--|
--| for > operator
--|
with function ">" (Left  : in Data_Type;
                   Right : in     Data_Type) return Boolean is <>;


package Simple_Bin_Tree
  is

  Bin_Tree_Full              : exception;
  Bin_Tree_Empty             : exception;
  Record_Not_Found_On_Delete : exception;

  type Iterator_Type is limited private;
  type Bin_Tree_Type is limited private;
  type Iterator_Kind_Type is (Pre_Order, In_Order, Post_Order);

  --|
  --|
  ------------------------------------------------------------------
  ------------------------------------------------------------------
  ------   B I N A R Y   T R E E    O P E R A T I O N S    ----------
  ------------------------------------------------------------------
  ------------------------------------------------------------------
  --|
  --|

  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |                 Add                 |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  procedure Add(Data            : in     Data_Type;
                In_The_Bin_Tree : in out Bin_Tree_Type);
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |              Remove                 |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  procedure Remove(The_Bin_Tree : in out Bin_Tree_Type;
                   Data         : in     Data_Type);
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |                Seek                 |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  procedure Seek(In_The_Bin_Tree : in     Bin_Tree_Type;
                 Data            : in     Data_Type;
                 Success         :    out Boolean;
                 Found_Record    :    out Data_Type);
  --|
  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |             Size_Of                 |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  function Size_Of(The_Bin_Tree : in     Bin_Tree_Type) return Integer;

  --|                      |_____________________________________|
  --|                      |                                     |
  --|                      |          Clear                      |
  --|                      |_____________________________________|
  --|                      |                                     |
  --|
  procedure Clear(The_Bin_Tree : in out Bin_Tree_Type);
  --|
  --|
  ------------------------------------------------------------------
  ------------------------------------------------------------------
  ------------------          I T E R A T O R     ------------------
  ------------------------------------------------------------------
  ------------------------------------------------------------------
  --|
  --|
```

```
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |              Initialize             |
--|                      |_____________________________________|
--|                      |                                     |
procedure Initialize(Iterator         :    out Iterator_Type;
                     For_The_Bin_Tree : in     Bin_Tree_Type;
                     Iterator_Kind    : in     Iterator_Kind_Type := In_Order);
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |              Is_Done                |
--|                      |_____________________________________|
--|                      |                                     |
--|
function Is_Done(Iterator : in     Iterator_Type) return Boolean;
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |      Current_Value                  |
--|                      |_____________________________________|
--|                      |                                     |
--|
function Current_Value(Iterator : in     Iterator_Type) return Data_Type;
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |      To_Next_Value                  |
--|                      |_____________________________________|
--|                      |                                     |
--|
procedure To_Next_Value(Iterator : in out Iterator_Type);
--|
--|
-------------------------------------------------------------------
-------------------------------------------------------------------
------------------   S T A T I S T I C  S    ----------------------
-------------------------------------------------------------------
-------------------------------------------------------------------
--|
--|

--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |       Print_Statistics              |
--|                      |_____________________________________|
--|                      |                                     |
--|
procedure Print_Statistics(Bin_Tree : in     Bin_Tree_Type);
--|
--|
-------------------------------------------------------------------
-------------------------------------------------------------------
------------------  P R I V A T E   P A R T   --------------------
-------------------------------------------------------------------
-------------------------------------------------------------------
--|
--|
private
--|
--|  BIN_TREE table
--|
type Bin_Tree_Table_Array_Type is array (1 .. 2 ** Max_Number_Of_Items - 1) of Data_Type;
--|
--|
--|
type Bin_Tree_Type
   is
   record
   Bin_Tree_Table_Array : aliased Bin_Tree_Table_Array_Type;
   How_Many             : Natural := 0;
end record;
--|
--|  needed for the iterator
--|
type Bin_Tree_Table_Array_Access_Type is access all Bin_Tree_Table_Array_Type;
--|
--|  iterator
--|
type Iterator_Type is
   record
   Iterator_Kind              : Iterator_Kind_Type              := In_Order;
   Current_Index              : Integer                         := 0;
   Previous_Index             : Integer                         := 0;
   Bin_Tree_Table_Array_Access : Bin_Tree_Table_Array_Access_Type := null;
end record;


type Statistics_Type is
   record
   Add_Calls    : Integer := 0;
   Delete_Calls : Integer := 0;
   Seek_Calls   : Integer := 0;
end record;
```

```
   Statistics      : Statistics_Type;
   Post_Order_Done : Boolean := False;

end Simple_Bin_Tree;
```

Fichier Simple_Bin_Tree.adb :

```
with Ada.Text_Io;

package body Simple_Bin_Tree
   is
   --|
   --|
   ---------------------------------------------------------------------
   ---------------------------------------------------------------------
   ------   B I N A R Y   T R E E   O P E R A T I O N S   ----------
   ---------------------------------------------------------------------
   ---------------------------------------------------------------------
   --|
   --|

   --|
   --|                    |_____________________________________|
   --|                    |                                     |
   --|                    |                Add                  |
   --|                    |_____________________________________|
   --|                    |                                     |
   --|
   procedure Add(Data            : in     Data_Type;
                 In_The_Bin_Tree : in out Bin_Tree_Type)
      is
      Index : Positive := 1;
      begin

      while not (Null_Data = In_The_Bin_Tree.Bin_Tree_Table_Array(Index)) loop

         if Data > In_The_Bin_Tree.Bin_Tree_Table_Array(Index)
            then
            Index := 2 * Index + 1;
            else
            Index := 2 * Index;
         end if;

      end loop;

      In_The_Bin_Tree.Bin_Tree_Table_Array(Index) := Data;
      In_The_Bin_Tree.How_Many                    := In_The_Bin_Tree.How_Many + 1;
      exception
      when Constraint_Error =>
      Ada.Text_Io.Put("add" & Integer'Image(Index) & Integer'Image(In_The_Bin_Tree.How_Many));
   end Add;
   --|
   --|                    |_____________________________________|
   --|                    |                                     |
   --|                    |             Remove                  |
   --|                    |_____________________________________|
   --|                    |                                     |
   --|
   procedure Remove(The_Bin_Tree : in out Bin_Tree_Type;
                    Data         : in     Data_Type)
      is

      --
      --               There are several cases to consider:
      --
      --1) Deleting a leaf: Deleting a node with no children is easy,
      --   as we can simply remove it from the tree.
      --
      --2) Deleting a node with one child: Delete it and replace it with its child.
      --
      --3) Deleting a node with two children:
      --   Suppose the node to be deleted is called Z.
      --   We replace the value of Z with either its in-order successor
      --   (the left-most child of the right subtree)
      --   or the in-order predecessor (the right-most child of the left subtree).
      --   Once we find either the in-order successor or predecessor,
      --   swap it with Z, and then delete it.
      --   Since either of these nodes must have less than two children
      --   (otherwise it cannot be the in-order successor or predecessor),
      --   it can be deleted using the previous two cases.
      --
      --  In a good implementation, it is generally recommended
      --  to avoid consistently using one of these nodes,
      --  because this can unbalance the tree  (to be done).

      --|
      --|                    |_____________________________________|
      --|                    |                                     |
      --|                    |      Delete_A_Node                  |
      --|                    |_____________________________________|
```

```
--|                    |                               |
--|
procedure Delete_A_Node(Index_Of_Node_To_Be_Deleted : in     Positive)
   is

   type Table_Type is array (1 .. 2 ** Max_Number_Of_Items - 1) of Natural;
   --|
   --|                    |_____________________________________|
   --|                    |                                     |
   --|                    |   Attach_Subtree_To_Parent          |
   --|                    |_____________________________________|
   --|                    |                                     |
   --|
   procedure Attach_Subtree_To_Parent(Index_Of_Current : in     Positive)
      is
      Index                     : Natural    := Index_Of_Current;
      Table_Index               : Positive   := Positive'First;
      Table                     : Table_Type := (others => 0);
      procedure Fill_Table(Index : in     Positive)
         is
         begin
         --|
         --| end of branch  ?    go back up
         --|
         if Null_Data = The_Bin_Tree.Bin_Tree_Table_Array(Index)
            then
            return;
         end if;
         --|
         --| LOAD INDEX PRE ORDER !!!!!
         --|
         Table(Table_Index) := Index;
         Table_Index        := 1 + Table_Index;
         --|
         --|  none of the above : keep walking the tree (customary dual recursion)
         --|
         Fill_Table(Index => 2 * Index);
         Fill_Table(Index => 2 * Index + 1);

      end Fill_Table;
      Subtract_Table : Table_Type := (others => 0);
      Dummy_Index    : Positive   := Positive'First;
      Sub_Value      : Positive   := Positive'First;

      begin
      --|
      --|  create a table to change the indexes in the tree by substraction
      --|
      Dummy_Index := 6;
      Sub_Value   := 2;

      for I in 3 .. Subtract_Table'Last loop
         if I = Dummy_Index
            then
            Sub_Value   := 2 * Sub_Value;
            Dummy_Index := 2 * Dummy_Index;
         end if;
         Subtract_Table(I) := Sub_Value;
      end loop;

      Table := (others => 0);
      Fill_Table(Index => Index_Of_Current);

      --              Ada.Text_Io.Put_Line("Fist One, up by / 2    Table(2)"
      --                                   & Integer'Image(Table(2) / 2)
      --                                   & " <="
      --                                   & Integer'Image(Table(2)));
      --|
      --|  care for the first one :: go up one step :=>  /2
      --|
      The_Bin_Tree.Bin_Tree_Table_Array(Table(2) / 2) :=
      The_Bin_Tree.Bin_Tree_Table_Array(Table(2));
      The_Bin_Tree.Bin_Tree_Table_Array(Table(2)) := Null_Data;
      --|
      --|  care for the others using substraction table
      --|
      for I in 3 .. Table_Index - 1 loop
         if Table(I) /= 0
            then
            --                    Ada.Text_Io.Put_Line("I"
            --                                         & Integer'Image(I)
            --                                         & " Table(I)"
            -- & Integer'Image(Table(I) - Subtract_Table(Table(I)))
            --                                         & " <="
            --                                         & Integer'Image(Table(I)));

            The_Bin_Tree.Bin_Tree_Table_Array(Table(I) - Subtract_Table(Table(I))) :=
            The_Bin_Tree.Bin_Tree_Table_Array(Table(I));
            The_Bin_Tree.Bin_Tree_Table_Array(Table(I)) := Null_Data;
         end if;
      end loop;

   end Attach_Subtree_To_Parent;

   Index_Of_Successor : Positive;
   Index_Of_Current   : Positive := Index_Of_Node_To_Be_Deleted;
   begin
```

```
--Ada.Text_Io.Put_Line("Index_Of_Current"
--& Integer'Image(Index_Of_Current));
--|
--|=========================================
--| The Node to be deleted has no children
--|=========================================
--|
if The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2 + 1) = Null_Data
   and then The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2) = Null_Data
   then
   --Ada.Text_Io.Put_Line("The Node to be deleted has no children");

   The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current) := Null_Data;
   --|
   --|=======================================================
   --| The Node To Be Deleted Has A Right Child But No Left Child
   --|=======================================================
   --|
   elsif The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2 + 1) /= Null_Data
   and then The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2) = Null_Data
   then
   --Ada.Text_Io.Put_Line("The Node To Be Deleted Has A Right Child But No Left Child");
   --|
   --|Attach right subtree To Parent
   --|
   Attach_Subtree_To_Parent(Index_Of_Current => Index_Of_Current);
   --|
   --|=======================================================
   --| The Node To Be Deleted Has A Left Child But No Right Child
   --|=======================================================
   --|
   elsif The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2 + 1) = Null_Data
   and then The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2) /= Null_Data
   then
   --Ada.Text_Io.Put_Line("The Node To Be Deleted Has A Left Child But No Right Child");
   --|
   --|Attach left subtree To Parent
   --|
   Attach_Subtree_To_Parent(Index_Of_Current => Index_Of_Current);
   --|
   --|=========================================
   --| The Node To Be Deleted Has Two Children
   --|=========================================
   --|
   elsif The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2 + 1) /= Null_Data
   and then The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2) /= Null_Data
   then
   --Ada.Text_Io.Put_Line("The Node To Be Deleted Has Two Children");


   --3) Deleting a node with two children:
   --   Suppose the node to be deleted is called Z.
   --   We replace the value of Z with either its in-order successor
   --   (the left-most child of the right subtree)
   --   or the in-order predecessor (the right-most child of the left subtree).
   --   Once we find either the in-order successor or predecessor,
   --   swap it with Z, and then delete it.
   --   Since either of these nodes must have less than two children
   --   (otherwise it cannot be the in-order successor or predecessor),
   --   it can be deleted using the previous two cases.
   --
   --  In a good implementation, it is generally recommended
   --  to avoid consistently using one of these nodes,
   --  because this can unbalance the tree  (to be done).

   --|
   --| get inorder successor  :  the left-most child of the right subtree
   --|

   --|
   --| 1) get to the right subtree
   --|
   Index_Of_Current := 2 * Index_Of_Current + 1;
   --|
   --| 2) find out the leftmost child: it has no left child !
   --|
   while not (The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current) /= Null_Data) loop
      Index_Of_Current := 2 * Index_Of_Current;
   end loop;
   --|
   --| swap  both
   --|
   The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Node_To_Be_Deleted)
   := The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current);
   --|
   --| two cases left: no child at all or a right child
   --|
   if The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2 + 1) = Null_Data
      and then The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current * 2) = Null_Data
      then
      --|
      --|no child clear node
      --|
      The_Bin_Tree.Bin_Tree_Table_Array(Index_Of_Current) := Null_Data;
      else
      --|
      --|a righ child : Attach right subtree To  PARENT =====>>> (Index_Of_Current/2)
```

```
            --|
            Attach_Subtree_To_Parent(Index_Of_Current => Index_Of_Current / 2);
         end if;
      end if;
   end Delete_A_Node;

   Found : Boolean  := False;
   Index : Positive := 1;
   --|
   --| find the node to be removed
   --|
   begin

   while not (Null_Data = The_Bin_Tree.Bin_Tree_Table_Array(Index)) loop

      if Data > The_Bin_Tree.Bin_Tree_Table_Array(Index)
         then
         Index := 2 * Index + 1;
         elsif Data = The_Bin_Tree.Bin_Tree_Table_Array(Index)
         then
         Found := True;
         exit;
         else
         Index := 2 * Index;
      end if;
   end loop;
   --|
   --| ok remove else raise error
   --|
   if Found
      then
      Delete_A_Node(Index_Of_Node_To_Be_Deleted => Index);
      else
      raise Record_Not_Found_On_Delete;
   end if;
   --|
   --| keep track
   --|
   The_Bin_Tree.How_Many := The_Bin_Tree.How_Many - 1;

end Remove;
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |                Seek                 |
--|                      |_____________________________________|
--|                      |                                     |
--|
procedure Seek(In_The_Bin_Tree : in     Bin_Tree_Type;
               Data            : in     Data_Type;
               Success         :    out Boolean;
               Found_Record    :    out Data_Type)
   is

   Index : Positive := 1;
   begin

   while not (Null_Data = In_The_Bin_Tree.Bin_Tree_Table_Array(Index)) loop

      if Data > In_The_Bin_Tree.Bin_Tree_Table_Array(Index)
         then
         Index := 2 * Index + 1;

         elsif Data = In_The_Bin_Tree.Bin_Tree_Table_Array(Index)
         then
         Success      := True;
         Found_Record := In_The_Bin_Tree.Bin_Tree_Table_Array(Index);
         return;
         else
         Index := 2 * Index;
      end if;

   end loop;

   Success := False;
   exception
   when Constraint_Error =>
   Ada.Text_Io.Put(Integer'Image(Index));
   Success := False;
end Seek;
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |             Size_Of                 |
--|                      |_____________________________________|
--|                      |                                     |
--|
function Size_Of(The_Bin_Tree : in     Bin_Tree_Type) return Integer
   is
   begin
   return The_Bin_Tree.How_Many;
end Size_Of;
--|                      |_____________________________________|
--|                      |                                     |
--|                      |          Clear                      |
--|                      |_____________________________________|
--|                      |                                     |
```

```
--|
procedure Clear(The_Bin_Tree : in out Bin_Tree_Type)
   is
   begin
   The_Bin_Tree.Bin_Tree_Table_Array := (others => Null_Data);
   The_Bin_Tree.How_Many             := 0;
end Clear;
--|
--|
-----------------------------------------------------------------
-----------------------------------------------------------------
------------------          I T E R A T O R     ------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
--|
--|

--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |            Initialize               |
--|                      |_____________________________________|
--|                      |                                     |
--|
procedure Initialize(Iterator         :    out Iterator_Type;
                     For_The_Bin_Tree : in     Bin_Tree_Type;
                     Iterator_Kind    : in     Iterator_Kind_Type := In_Order)
   is

   begin
   Iterator := (Iterator_Kind               => Iterator_Kind,
                Previous_Index              => 0,
                Current_Index               => 1,
                Bin_Tree_Table_Array_Access =>
                new Bin_Tree_Table_Array_Type'(For_The_Bin_Tree.Bin_Tree_Table_Array));
   --|
   --| get first value if any
   --|
   Post_Order_Done := False;
   To_Next_Value(Iterator => Iterator);

end Initialize;
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |            Is_Done                  |
--|                      |_____________________________________|
--|                      |                                     |
--|
function Is_Done(Iterator : in     Iterator_Type) return Boolean
   is
   begin

   if Post_Order = Iterator.Iterator_Kind
      then
      if Post_Order_Done
         then
         return True;
      end if;
      Post_Order_Done := Iterator.Current_Index = 0
      and then Iterator.Previous_Index = 1;
      return False;

      else
      return Iterator.Current_Index = 0
      and then Iterator.Previous_Index = 1;
   end if;
end Is_Done;
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |      Current_Value                  |
--|                      |_____________________________________|
--|                      |                                     |
--|
function Current_Value(Iterator : in     Iterator_Type) return Data_Type
   is
   begin
   return Iterator.Bin_Tree_Table_Array_Access.all (Iterator.Previous_Index);
end Current_Value;
--|
--|                      |_____________________________________|
--|                      |                                     |
--|                      |      To_Next_Value                  |
--|                      |_____________________________________|
--|                      |                                     |
--|
procedure To_Next_Value(Iterator : in out Iterator_Type)
   is
   begin
   --|
   --|upon entry to a node they are several possibilities:
   --|
   --|1) the position is void : go up by dividing by 2
   --|
   --|2) it is the first time we arrive in this node (current/2 = previous  )   go left
   --|
```

```
      --|3) it is the second time we arrive in this node as we are returning from the left
      --|   (current *2 = previous  )   go right
      --|
      --|4) it is the third time we arrive in this node as we are returning from the right
      --|   (current *2 +1 = previous  )   go up
      --|
      while not (Iterator.Current_Index = 0 and then Iterator.Previous_Index = 1) loop
         --|
         --|1) the position is void : go up by dividing by 2
         --|
         if Null_Data = Iterator.Bin_Tree_Table_Array_Access.all (Iterator.Current_Index)
            then
            Iterator.Previous_Index := Iterator.Current_Index;
            Iterator.Current_Index  := Iterator.Previous_Index / 2;
            --|
            --|2) it is the first time we arrive in this node (current/2 = previous)   go left
            --|
            elsif Iterator.Current_Index / 2 = Iterator.Previous_Index
            then
            Iterator.Previous_Index := Iterator.Current_Index;
            Iterator.Current_Index  := Iterator.Previous_Index * 2;

            if Pre_Order = Iterator.Iterator_Kind
               then
               return;
            end if;
            --|
            --|3) it is the second time we arrive in this node as we are returning from the left
            --|   (current *2 = previous  )   go right
            --|
            elsif Iterator.Current_Index * 2 = Iterator.Previous_Index
            then
            Iterator.Previous_Index := Iterator.Current_Index;
            Iterator.Current_Index  := Iterator.Previous_Index * 2 + 1;

            if In_Order = Iterator.Iterator_Kind
               then
               return;
            end if;
            --|
            --|4) it is the third time we arrive in this node as we are returning from the right
            --|   (current *2 +1 = previous  )   go up
            --|
            elsif Iterator.Current_Index * 2 + 1 = Iterator.Previous_Index
            then
            Iterator.Previous_Index := Iterator.Current_Index;
            Iterator.Current_Index  := Iterator.Previous_Index / 2;

            if Post_Order = Iterator.Iterator_Kind
               then
               return;
            end if;
         end if;
      end loop;

   end To_Next_Value;
   --|
   --|
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   ------------------   S T A T I S T I C  S    ---------------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   --|
   --|

   --|
   --|                     |_____________________________|
   --|                     |                             |
   --|                     |       Print_Statistics      |
   --|                     |_____________________________|
   --|                     |                             |
   --|
   procedure Print_Statistics(Bin_Tree : in     Bin_Tree_Type)
      is
      begin
      null;
   end Print_Statistics;


end Simple_Bin_Tree;
```

14.10 LE CODE DE L'ARBRE BINAIRE (POINTEURS)

Fichier linked_bin_tree.ads :

```
with Ada.Unchecked_Deallocation;
with Linked_Stack;
```

```
generic

type Data_Type is private;
--|
--| for = operator
--|
with function "=" (Left  : in Data_Type;
                   Right : in     Data_Type) return Boolean is <>;
--|
--| for > operator
--|
with function ">" (Left  : in Data_Type;
                   Right : in     Data_Type) return Boolean is <>;

package Linked_Bin_Tree
  is

  Bin_Tree_Empty             : exception;
  Record_Not_Found_On_Delete : exception;

  type Iterator_Type is limited private;
  type Bin_Tree_Type is limited private;
  type Iterator_Kind_Type is (Pre_Order, In_Order, Post_Order);
  --|
  --|              |-------|
  --|              | Clear |
  --|              |_______|
  --|
  procedure Clear(The_Bin_Tree : in out Bin_Tree_Type);
  --|
  --|              |-------------------------------|
  --|              |           Insert              |
  --|              |_______________________________|
  --|
  procedure Insert(The_Data        : in     Data_Type;
                   In_The_Bin_Tree : in out Bin_Tree_Type);
  --|
  --|              |-------------------------------|
  --|              |            Delete             |
  --|              |_______________________________|
  --|
  procedure Delete(The_Data        : in     Data_Type;
                   In_The_Bin_Tree : in out Bin_Tree_Type);
  --|
  --|              |-------------------------------|
  --|              |        Size_Of                |
  --|              |_______________________________|
  --|
  function Size_Of(The_Bin_Tree : in     Bin_Tree_Type) return Integer;
  --|
  --|              |_______________________________|
  --|              |                               |
  --|              |               Seek            |
  --|              |_______________________________|
  --|              |                               |
  --|
  procedure Seek(In_The_Bin_Tree : in     Bin_Tree_Type;
                 Data            : in     Data_Type;
                 Success         :    out Boolean;
                 Found_Record    :    out Data_Type);

  ------------------------------------------------------------------
  ------------------------------------------------------------------
  -------------------          I T E R A T O R     ------------------
  ------------------------------------------------------------------
  ------------------------------------------------------------------
  --|
  --|

  --|
  --|              |_______________________________|
  --|              |                               |
  --|              |            Initialize         |
  --|              |_______________________________|
  --|              |                               |
  --|
  procedure Initialize(Iterator         :    out Iterator_Type;
                       For_The_Bin_Tree : in     Bin_Tree_Type;
                       Iterator_Kind    : in     Iterator_Kind_Type := In_Order);
  --|
  --|              |_______________________________|
  --|              |                               |
  --|              |           Is_Done             |
  --|              |_______________________________|
  --|              |                               |
  --|
  function Is_Done(Iterator : in     Iterator_Type) return Boolean;
  --|
  --|              |_______________________________|
  --|              |                               |
  --|              |      Current_Value            |
  --|              |_______________________________|
  --|              |                               |
  --|
  function Current_Value(Iterator : in     Iterator_Type) return Data_Type;
  --|
  --|              |_______________________________|
```

```
   --|              |                              |
   --|              |      To_Next_Value           |
   --|              |______________________________|
   --|              |                              |
   --|
   procedure To_Next_Value(Iterator : in out Iterator_Type);
   --|
   --|
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   ------------------   S T A T I S T I C  S    ---------------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   --|
   --|

   --|
   --|              |______________________________|
   --|              |                              |
   --|              |       Print_Statistics       |
   --|              |______________________________|
   --|              |                              |
   --|
   procedure Print_Statistics(Bin_Tree : in     Bin_Tree_Type);
   --|
   --|
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   ------------------       P R I V A T E       ---------------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   --|
   --|
   private
   --|
   --|
   --|
   type Node_Type;
   --|
   --|
   --|
   type Node_Access_Type is access all Node_Type;
   --|
   --|
   --|
   type Bin_Tree_Type is
      record
      Access_To_Top   : Node_Access_Type := null;
      How_Many_Stored : Integer          := 0;
   end record;
   --|
   --|
   --|
   type Node_Type is
      record
      Data  : Data_Type;
      Left  : Node_Access_Type := null;
      Right : Node_Access_Type := null;
   end record;
   --|
   --|
   --|
   package My_Stack is new Linked_Stack(Data_Type => Node_Access_Type);
   Iterator_Stack : My_Stack.Stack_Type;

   --|
   --| general data
   --|
   type Iterator_Type is
      record
      Current_Node_Access : Node_Access_Type   := null;
      Iterator_Kind       : Iterator_Kind_Type := In_Order;
      First_Time          : Boolean            := True;
      How_Many_Stored     : Natural            := 0;
      Current_Count       : Natural            := 0;
      On_Off              : Boolean            := True;
      Loop_Done           : Boolean            := False;
      Q                   : Node_Access_Type   := null;
   end record;
   --|
   --|  used in deletions
   --|
   procedure Free_Node is new Ada.Unchecked_Deallocation(Object => Node_Type,
                                                        Name   => Node_Access_Type);

end Linked_Bin_Tree;
```

Fichier linked_bin_tree.adb :

```
with Ada.Text_Io;

package body Linked_Bin_Tree
  is
```

```
--|
--|              |-------|
--|              | Clear |
--|              |_______|
--|
procedure Clear(The_Bin_Tree : in out Bin_Tree_Type)
   is
   --
   -- The procedure "Destroy_Tree" goes to the bottom of each part of the tree,
   -- that is, searching while there is a non-null node,
   -- deletes that leaf, and then it works its way back up.
   -- The procedure deletes the leftmost node, then the right child node
   -- from the leftmost node's parent node,
   -- then it deletes the parent node, then works its way back to deleting
   -- the other child node of the parent of the node it just deleted,
   -- and it continues this deletion working its way up to the node
   -- of the tree upon which delete_tree was originally called.
   --
   --
   procedure Destroy_Tree(Link_To_Current_Node : in out Node_Access_Type)
      is
      begin
      --|
      --| end of branch  ?    go back up
      --|
      if null = Link_To_Current_Node
         then
         return;
      end if;
      --|
      --|  none of the above : keep walking the tree (customary dual recursion)
      --|
      Destroy_Tree(Link_To_Current_Node => Link_To_Current_Node.Left);
      Destroy_Tree(Link_To_Current_Node => Link_To_Current_Node.Right);
      --|
      --| clear leaf
      --|
      Free_Node(X => Link_To_Current_Node);
   end Destroy_Tree;

   begin
   --|
   --| clear tree
   --|
   Destroy_Tree(Link_To_Current_Node => The_Bin_Tree.Access_To_Top);
   --|
   --| clear number
   --|
   The_Bin_Tree.How_Many_Stored := 0;
end Clear;
--|
--|              |--------------------------------|
--|              |           Insert               |
--|              |________________________________|
--|
procedure Insert(The_Data        : in     Data_Type;
                 In_The_Bin_Tree : in out Bin_Tree_Type)
   is
   type Right_Left_Type is (R, L);
   Right_Left            : Right_Left_Type  := Right_Left_Type'First;
   Link_To_Previous_Node : Node_Access_Type := In_The_Bin_Tree.Access_To_Top;
   Link_To_Current_Node  : Node_Access_Type := In_The_Bin_Tree.Access_To_Top;
   begin
   --|
   --| first node ??
   --|
   if null = Link_To_Current_Node
      then
      In_The_Bin_Tree.Access_To_Top := new Node_Type'(Left  => null,
                                                      Data  => The_Data,
                                                      Right => null);
      else
      --|
      --| not the first to be added: seek position to insert
      --|
      while null /= Link_To_Current_Node loop
         --|
         --| keep previous    S O   T H A T   Y O U   C A N   A D D !!!!
         --|
         Link_To_Previous_Node := Link_To_Current_Node;
         --|
         --| left or right
         --|
         if Link_To_Current_Node.Data > The_Data
            then

            Right_Left           := L;
            Link_To_Current_Node := Link_To_Current_Node.Left;
            else

            Link_To_Current_Node := Link_To_Current_Node.Right;
            Right_Left           := R;
         end if;
      end loop;
      --|
      --| position found do the insert
      --|
```

```
      if Right_Left = L
         then
         Link_To_Previous_Node.Left := new Node_Type'(Left  => null,
                                                      Data  => The_Data,
                                                      Right => null);
         else
         Link_To_Previous_Node.Right := new Node_Type'(Left  => null,
                                                       Data  => The_Data,
                                                       Right => null);
      end if;
   end if;
   --|
   --| keep track
   --|
   In_The_Bin_Tree.How_Many_Stored := 1 + In_The_Bin_Tree.How_Many_Stored;
end Insert;
--|
--|              |------------------------------|
--|              |            Delete            |
--|              |______________________________|
--|
procedure Delete(The_Data        : in     Data_Type;
                 In_The_Bin_Tree : in out Bin_Tree_Type)
   is
   --
   --There are several cases to consider:
   --
   --1) Deleting a leaf: Deleting a node with no children is easy,
   --   as we can simply remove it from the tree.
   --
   --2) Deleting a node with one child: Delete it and replace it with its child.
   --
   --3) Deleting a node with two children:
   --   Suppose the node to be deleted is called Z.
   --   We replace the value of Z with either its in-order successor
   --   (the left-most child of the right subtree)
   --   or the in-order predecessor (the right-most child of the left subtree).
   --   Once we find either the in-order successor or predecessor,
   --   swap it with Z, and then delete it.
   --   Since either of these nodes must have less than two children
   --   (otherwise it cannot be the in-order successor or predecessor),
   --   it can be deleted using the previous two cases.
   --
   --  In a good implementation, it is generally recommended
   --  to avoid consistently using one of these nodes,
   --  because this can unbalance the tree  (to be done).

   type Right_Left_Type is (R, L);

   procedure Delete_A_Node(Node_To_Be_Deleted     : in out Node_Access_Type;
                           Parent_Of_The_Node_To_Be_Deleted        : in out Node_Access_Type;
                           Side_From_The_Parent_To_The_Node_To_Be_Deleted : in out Right_Left_Type)
      is
      Successor : Node_Access_Type;
      Current   : Node_Access_Type := Node_To_Be_Deleted;
      begin
      --|
      --|============================================
      --| The Node to be deleted has no children
      --|============================================
      --|
      if Current.Right = null
         and then Current.Left = null
         then
         --|
         --|Clear the Appropriate Side of Parent
         --|
         if Side_From_The_Parent_To_The_Node_To_Be_Deleted = L
            then
            Free_Node(X => Parent_Of_The_Node_To_Be_Deleted.Left);
            else
            Free_Node(X => Parent_Of_The_Node_To_Be_Deleted.Right);
         end if;
         --|
         --|============================================================
         --| The Node To Be Deleted Has A Right Child But No Left Child
         --|============================================================
         --|
         elsif Current.Right /= null
         and then Current.Left = null
         then
         Successor := Current.Right;
         --|
         --|Attach Node To Parent
         --|
         if Side_From_The_Parent_To_The_Node_To_Be_Deleted = L
            then
            Parent_Of_The_Node_To_Be_Deleted.Left := Successor;
            else
            Parent_Of_The_Node_To_Be_Deleted.Right := Successor;
         end if;

         Free_Node(X => Current);
         --|
         --|============================================================
         --| The Node To Be Deleted Has A Left Child But No Right Child
         --|============================================================
```

```
      --|
      elsif Current.Right = null
      and then Current.Left /= null
      then
      Successor := Current.Left;
      --|
      --|Attach Node To Parent
      --|
      if Side_From_The_Parent_To_The_Node_To_Be_Deleted = L
         then
         Parent_Of_The_Node_To_Be_Deleted.Left := Successor;
         else
         Parent_Of_The_Node_To_Be_Deleted.Right := Successor;
      end if;

      Free_Node(X => Current);
      --|
      --|=============================================
      --| Node To Be Deleted Has Two Children
      --|=============================================
      --|
      elsif Current.Right /= null
      and then Current.Left /= null
      then
      --|
      --| Set Current To The Left Child of The Node To Be Deleted
      --|
      Current := Current.Left;
      if Current.Right /= null
         then
         Successor := Current.Right;
         --|
         --| walk Down To The Right end of The Subtree of The Left Child
         --| of The Node To Be Deleted
         --|
         while Current.Right /= null loop
            Current   := Successor;
            Successor := Current.Right;
         end loop;
         --|
         --| replace The Node To Be Deleted with The Node Found
         --| in this case :
         --| Attach left Child of Node To Be Deleted
         --|
         Current.Right   := null;
         Successor.Left  := Node_To_Be_Deleted.Left;
         Successor.Right := Node_To_Be_Deleted.Right;
         --|
         --| Attach Node To Parent
         --|
         if Side_From_The_Parent_To_The_Node_To_Be_Deleted = L
            then
            Parent_Of_The_Node_To_Be_Deleted.Left := Successor;
            else
            Parent_Of_The_Node_To_Be_Deleted.Right := Successor;
         end if;
         --|
         --| There is No Right Subtree because the right pointer is null :
         --| Replace The Node To Be Deleted with Its Left Child
         --| in this case :
         --| Attach Right Child of Node To Be Deleted
         --|
         else
         Successor       := Current;
         Successor.Right := Node_To_Be_Deleted.Right;
         --|
         --| Attach Node To Parent
         --|
         if Side_From_The_Parent_To_The_Node_To_Be_Deleted = L
            then
            Parent_Of_The_Node_To_Be_Deleted.Left := Successor;
            else
            Parent_Of_The_Node_To_Be_Deleted.Right := Successor;
         end if;
         Free_Node(X => Node_To_Be_Deleted);
      end if;
   end if;
end Delete_A_Node;


Right_Left                 : Right_Left_Type  := Right_Left_Type'First;
Parent_Node                : Node_Access_Type := In_The_Bin_Tree.Access_To_Top;
Current_Node               : Node_Access_Type := In_The_Bin_Tree.Access_To_Top;
Found                      : Boolean          := False;
Side_From_Parent_To_Current : Right_Left_Type  := Right_Left_Type'First;

begin
--|
--| find the node to be removed
--|
while not (null = Current_Node
           or else Found) loop

   if Current_Node.Data = The_Data
      then
      Found := True;
```

```
         elsif The_Data > Current_Node.Data
         then
         Side_From_Parent_To_Current := R;
         Parent_Node                 := Current_Node;
         Current_Node                := Current_Node.Right;

         elsif Current_Node.Data > The_Data
         then
         Side_From_Parent_To_Current := L;
         Parent_Node                 := Current_Node;
         Current_Node                := Current_Node.Left;
      end if;
   end loop;
   --|
   --| ok remove else raise error
   --|
   if Found
      then
      Delete_A_Node(Node_To_Be_Deleted                           => Current_Node,
                    Parent_Of_The_Node_To_Be_Deleted             => Parent_Node,
                    Side_From_The_Parent_To_The_Node_To_Be_Deleted => Side_From_Parent_To_Current);
      else
      raise Record_Not_Found_On_Delete;
   end if;
   --|
   --| keep track
   --|
   In_The_Bin_Tree.How_Many_Stored := In_The_Bin_Tree.How_Many_Stored - 1;

end Delete;
--|
--|              |-------------------------------|
--|              |        Size_Of                |
--|              |_______________________________|
--|
function Size_Of(The_Bin_Tree : in     Bin_Tree_Type)
   return Integer
   is
   begin
   return The_Bin_Tree.How_Many_Stored;
end Size_Of;
--|
--|              |_______________________________|
--|              |                               |
--|              |               Seek            |
--|              |_______________________________|
--|              |                               |
--|
procedure Seek(In_The_Bin_Tree : in     Bin_Tree_Type;
               Data            : in     Data_Type;
               Success         :    out Boolean;
               Found_Record    :    out Data_Type)
   is
   Link_To_Current_Node : Node_Access_Type := In_The_Bin_Tree.Access_To_Top;
   begin
   --|
   --| kepp looking
   --|
   while null /= Link_To_Current_Node loop

      if Link_To_Current_Node.Data = Data
         then
         Found_Record := Link_To_Current_Node.Data;
         Success      := True;
         return;

         elsif Link_To_Current_Node.Data > Data
         then
         Link_To_Current_Node := Link_To_Current_Node.Left;

         else
         Link_To_Current_Node := Link_To_Current_Node.Right;
      end if;
   end loop;
   --|
   --| not found
   --|
   Success := False;
end Seek;
--|
--|
-----------------------------------------------------------------
-----------------------------------------------------------------
-------------------          I T E R A T O R     ----------------
-----------------------------------------------------------------
-----------------------------------------------------------------
--|
--|

--|
--|              |_______________________________|
--|              |                               |
--|              |            Initialize         |
--|              |_______________________________|
--|              |                               |
--|
procedure Initialize(Iterator         :    out Iterator_Type;
```

```
                    For_The_Bin_Tree : in     Bin_Tree_Type;
                    Iterator_Kind    : in     Iterator_Kind_Type := In_Order)
   is
   begin
   --|
   --| reset and initialize
   --|
   Iterator := (Current_Node_Access => For_The_Bin_Tree.Access_To_Top,
                Iterator_Kind       => Iterator_Kind,
                First_Time          => True,
                How_Many_Stored     => For_The_Bin_Tree.How_Many_Stored,
                Current_Count       => 0,
                On_Off              => True,
                Loop_Done           => False,
                Q                   => null);
   --|
   --| clear stack
   --|
   My_Stack.Initialize(The_Stack => Iterator_Stack);
   --|
   --| get first data
   --|
   To_Next_Value(Iterator => Iterator);

end Initialize;
--|
--|               |_____________________________________|
--|               |                                     |
--|               |            Is_Done                  |
--|               |_____________________________________|
--|               |                                     |
--|
function Is_Done(Iterator : in     Iterator_Type)
   return Boolean
   is
   begin
   return Iterator.Current_Count = 1 + Iterator.How_Many_Stored;
end Is_Done;
--|
--|               |_____________________________________|
--|               |                                     |
--|               |      Current_Value                  |
--|               |_____________________________________|
--|               |                                     |
--|
function Current_Value(Iterator : in     Iterator_Type)
   return Data_Type
   is
   begin
   return Iterator.Current_Node_Access.Data;
end Current_Value;
--|
--|               |_____________________________________|
--|               |                                     |
--|               |      To_Next_Value                  |
--|               |_____________________________________|
--|               |                                     |
--|
procedure To_Next_Value(Iterator : in out Iterator_Type)
   is
   begin
   --|
   --| three iterators  select the one needed
   --|
   case Iterator.Iterator_Kind
      is
      --|
      --|=============================================
      --| P R E   O R D E R   I T E R A T O R
      --|=============================================
      --|
      when Pre_Order =>
      --
      -- D. Knuth the art of computer programming Volume 1 page 320 Addison Wesley
      --
      -- T1: set stack empty
      --     p <- top of tree
      -- T2: if p=null go to T4
      -- T3: visit pre order
      --     push p in stack
      --     p <- p.left
      --     go to T2
      -- T4: if stack empty algo is over
      --     else p<- top of stack
      --     pop stack
      -- T5: p <- P.Right
      --     go to T2
      --
      In_Pre_Order_Walk : declare
      begin
         --|
         --| loop to go back up if needed
         --|
         Main : loop
            --|
            --| not to be done the first time: we go to the deepest left immediately
            --|
```

```
      if Iterator.First_Time
         then
         Iterator.On_Off        := False;
         Iterator.Current_Count := 1 + Iterator.Current_Count;
         Iterator.First_Time    := False;
         exit Main;
      end if;
      --|
      --| the jump inside a loop is replaced by a test that's true 1/2 of the time
      --|
      while null /= Iterator.Current_Node_Access loop
         --|
         --| this time return and set flag
         --|
         if Iterator.On_Off
            then
            Iterator.Current_Count := 1 + Iterator.Current_Count;
            Iterator.On_Off        := False;
            exit Main;
         end if;
         --|
         --|set flag and push pointer
         --|
         Iterator.On_Off := True;
         My_Stack.Push(Data         => Iterator.Current_Node_Access,
                       In_The_Stack => Iterator_Stack);

         Iterator.Current_Node_Access := Iterator.Current_Node_Access.Left;
      end loop;
      --|
      --|retrieve pointer
      --|
      Iterator.Current_Node_Access := My_Stack.Top_Of(The_Stack => Iterator_Stack);
      --|
      --|remove last used and go right
      --|
      My_Stack.Pop(The_Stack => Iterator_Stack);
      Iterator.Current_Node_Access := Iterator.Current_Node_Access.Right;
   end loop Main;
   --|
   --|catch popping an empty stack (it also could be done by a test)
   --|
   exception
   when Constraint_Error =>
   Iterator.Current_Count := 1 + Iterator.Current_Count;
end In_Pre_Order_Walk;
--|
--|============================================
--| I N    O R D E R   I T E R A T O R
--|============================================
--|
when In_Order =>
--
-- D. Knuth the art of computer programming Volume 1 page 320 Addison Wesley
--
-- T1: set stack empty
--     p <- top of tree
-- T2: if p=null go to T4
-- T3: push p in stack
--     p <- p.left
--     go to T2
-- T4: if stack empty algo is over
--     else p<- top of stack
--     pop stack
-- T5: visit in order
--     p <- P.Right
--     go to T2
--          In_Order_Walk : declare
begin
   --|
   --| not to be done the first time: we go to the deepest left immediately
   --|
   if not Iterator.First_Time
      then
      Iterator.Current_Node_Access := Iterator.Current_Node_Access.Right;
   end if;
   --|
   --| go as far to the left as you can, pushing pointers in the stack
   --|
   while not (null = Iterator.Current_Node_Access) loop

      My_Stack.Push(Data         => Iterator.Current_Node_Access,
                    In_The_Stack => Iterator_Stack);

      Iterator.Current_Node_Access := Iterator.Current_Node_Access.Left;
   end loop;
   --|
   --| we reached an empty pointer get back one step
   --|
   Iterator.Current_Node_Access := My_Stack.Top_Of(The_Stack => Iterator_Stack);
   --|
   --| remove the last visited one
   --|
   My_Stack.Pop(The_Stack => Iterator_Stack);
   --|
   --|set flag so that we go to the right at the next call to To_Next_Value
   --|
```

```
            Iterator.First_Time    := False;
            Iterator.Current_Count := 1 + Iterator.Current_Count;
            --|
            --|catch popping an empty stack (it also could be done by a test)
            --|
            exception
            when Constraint_Error =>
            Iterator.Current_Count := 1 + Iterator.Current_Count;
         end In_Order_Walk;
         --|
         --|=============================================
         --| P O S T   O R D E R   I T E R A T O R
         --|=============================================
         --|
         when Post_Order =>
         --
         -- D. Knuth the art of computer programming Volume 1 page 565 Addison Wesley
         --
         -- T1: set stack empty
         --     p <- top of tree
         --     Q <- null
         -- T2: if p=null go to T4
         -- T3: push p in stack
         --     p <- p.left
         --     go to T2
         -- T4: if stack empty algo is over
         --     else p<- top of stack
         --     pop stack
         -- T5: if p.right = null or P.left =Q  got to T6
         --     p <- P.Right
         --     go to T2
         -- T6: visit post_order
         --     Q <- P
         --     go to T4
         --
         Post_Order_Walk : declare
         begin
            loop
               --|
               --| go as far to the left as you can, pushing pointers in the stack
               --|
               if Iterator.On_Off
                  then
                  while not (null = Iterator.Current_Node_Access) loop

                     My_Stack.Push(Data         => Iterator.Current_Node_Access,
                                   In_The_Stack => Iterator_Stack);

                     Iterator.Current_Node_Access := Iterator.Current_Node_Access.Left;
                  end loop;
               end if;
               --|
               --| we reached an empty pointer get back one step
               --|
               Iterator.Current_Node_Access := My_Stack.Top_Of(The_Stack => Iterator_Stack);
               --|
               --| back from right or right is null ????
               --|
               if (Iterator.Current_Node_Access.Right = null
                   or else Iterator.Current_Node_Access.Right = Iterator.Q)
                  then
                  --|
                  --| YES  keep Q for next time pop latest in stack and visit post_order
                  --|
                  Iterator.Q             := Iterator.Current_Node_Access;
                  Iterator.Current_Count := 1 + Iterator.Current_Count;
                  Iterator.On_Off        := False;
                  My_Stack.Pop(The_Stack => Iterator_Stack);
                  exit;
                  else
                  --|
                  --|NO use the loop to move in the tree
                  --|
                  Iterator.Current_Node_Access := Iterator.Current_Node_Access.Right;
                  Iterator.On_Off              := True;
               end if;
            end loop;
            --|
            --|catch popping an empty stack (it also could be done by a test)
            --|
            exception
            when Constraint_Error =>
            Iterator.Current_Count := 1 + Iterator.Current_Count;
         end Post_Order_Walk;
      end case;
   end To_Next_Value;
   --|
   --|
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   -------------------   S T A T I S T I C  S    --------------------
   ------------------------------------------------------------------
   ------------------------------------------------------------------
   --|
   --|

   --|
```

```
   --|              |_____________________________|
   --|              |                             |
   --|              |       Print_Statistics      |
   --|              |_____________________________|
   --|              |                             |
   --|
   procedure Print_Statistics(Bin_Tree : in     Bin_Tree_Type)
      is
      begin
      null;
   end Print_Statistics;

end Linked_Bin_Tree;
```

14.11 LE CODE DU TRI BULLE

Le code ci-dessous est un exemple de tri bulle. Il y a deux options pour l'écriture:

- Chaque tour de la boucle interne déplace l'élément le plus grand restant vers le haut (c'est le choix implémenté).
- Chaque tour de la boucle interne déplace l'élément le plus petit restant vers le bas.

A chaque début de la boucle interne, la taille du tableau restant à trier est diminuée de une case.
C'est un tri en N^2 à éviter absolument!

```
with Ada.Text_Io;
procedure S
   is
   subtype Item is Float range 0.0 .. 1000.0;

   type Table_Type is array (Integer range <>) of Item;

   procedure Bubble(Table : in out Table_Type)
      is
      Swap                     : Item;
L'attribut last est utilisé car on ne connaît pas à priori les limites du tableau;
      Last_Index_On_Second_Loop : Integer := Table'Last;
      begin
L'attribut "range" est utilisé car on ne connaît pas à priori les limites du tableau;
      for I in Table'Range loop
Diminution de la taille du tableau restant à trier l'attribut "pred" est utilisé car on ne connaît pas le type de l'index
         Last_Index_On_Second_Loop := Integer'Pred(Last_Index_On_Second_Loop);
L'attribut "first" est utilisé car on ne connaît pas à priori les limites du tableau;
         for J in Table'First .. Last_Index_On_Second_Loop loop
Si l'ordre n'est pas respecté, inverser les deux cases du tableau
            if Table(J + 1) < Table(J)
               then
               Swap         := Table(J);
               Table(J)     := Table(J + 1);
               Table(J + 1) := Swap;
            end if;
         end loop;
      end loop;
   end Bubble;
Données de test  (noter les limites du tableau!)
   T : Table_Type(3 .. 9) := (5.0, 6.0, 3.0, 11.0, 4.0, 20.0, 5.0 / 2.0);

   begin
Tri
   Bubble(Table => T);
Debug
   for I in T'Range loop
      Ada.Text_Io.Put_Line(Integer'Image(I) & Float'Image(T(I)));
   end loop;
end S;
```

14.12 LE CODE DU TRI EN TAS

```
--|
--|Daniel.Gaudry@wanadoo.fr
--|
--|                          H E A P   S O R T


with Ada.Text_Io;
--|
--|               |_________________|
--|               |                 |
--|               |     Exo_5       |
--|               |_________________|
--|               |                 |
--|
procedure Exo_5
   is

   subtype Data_Type is Integer  range 1 .. 100;
   subtype Index_Type is Integer range 1 .. 7;
   type A_Type is array (Index_Type'Range) of Data_Type;
```

```
A         : A_Type     := (7, 3, 5, 2, 4, 1, 6);
Heap_Size : Index_Type := A'Last;-- global to simplify reading !!!!
--|
--|               |_________________|
--|               |                 |
--|               | To_Screen       |
--|               |_________________|
--|               |                 |
--|
procedure To_Screen(A : in     A_Type)
   is
   begin

   for I in A'Range loop
      Ada.Text_Io.Put(Integer'Image(A(I)));
   end loop;

   Ada.Text_Io.New_Line;

end To_Screen;
--|
--|               |_________________|
--|               |                 |
--|               |    Swap         |
--|               |_________________|
--|               |                 |
--|
procedure Swap(A : in out A_Type;
               I : in     Index_Type;
               J : in     Index_Type)
   is
   Dummy : Data_Type;
   begin
   Dummy := A(I);
   A(I)  := A(J);
   A(J)  := Dummy;
end Swap;
--|
--|               |_________________|
--|               |                 |
--|               | Move_In_Heap    |
--|               |_________________|
--|               |                 |
--|
procedure Move_In_Heap(A : in out A_Type;
                       I : in     Index_Type)
   is
   Index_Of_Left_Child   : constant Integer := 2 * I;
   Index_Of_Right_Child  : constant Integer := Index_Of_Left_Child + 1;
   Index_Of_Largest_Item : Index_Type       := I;

   begin
   --|
   --| Keep index of largest (parent or left child)
   --|
   if Index_Of_Left_Child <= Heap_Size
      and then A(Index_Of_Left_Child) > A(I)
      then
      Index_Of_Largest_Item := Index_Of_Left_Child;
   end if;
   --|
   --|  Keep index of largest (parent or right child)
   --|
   if Index_Of_Right_Child <= Heap_Size
      and then A(Index_Of_Right_Child) > A(Index_Of_Largest_Item)
      then
      Index_Of_Largest_Item := Index_Of_Right_Child;
   end if;
   --|
   --|   D E B U G
   --|   ---------
   Ada.Text_Io.Put_Line("i"
                        & Integer'Image(I)
                        & " Index_Of_Left_Child"
                        & Integer'Image(Index_Of_Left_Child)
                        & " Index_Of_Right_Child"
                        & Integer'Image(Index_Of_Right_Child)
                        & " Index_Of_Largest_Item"
                        & Integer'Image(Index_Of_Largest_Item));

   To_Screen(A => A);
   --|
   --|     out of order  swap  and recursive call to check newly swapped parent
   --|
   if Index_Of_Largest_Item /= I
      then
      --|
      --|   D E B U G
      --|   ---------
      Ada.Text_Io.Put_Line("Swapping i"
                           & Integer'Image(I)
                           & " with "
                           & Integer'Image(Index_Of_Largest_Item));
      --|
      --|   swap
      --|
```

```
      Swap(A => A,
           I => I,
           J => Index_Of_Largest_Item);
      --|
      --|    recursive all
      --|
      Move_In_Heap(A => A,
                   I => Index_Of_Largest_Item);
   end if;

end Move_In_Heap;
--|
--|               |_________________|
--|               |                 |
--|               | Build_The_Heap  |
--|               |_________________|
--|               |                 |
--|
procedure Build_The_Heap(A : in out A_Type)
   is
   begin
   --|
   --|    start at the left on the one before the last row
   --|
   for I in reverse 1 .. A'Last / 2 loop
      --|
      --|   D E B U G
      --|   ---------
      Ada.Text_Io.Put_Line("Build_The_Heap with i ="
                           & Integer'Image(I));
      --|
      --|     move current parent if needed
      --|
      Move_In_Heap(A => A,
                   I => I);
   end loop;

end Build_The_Heap;
--|
--|               |_________________|
--|               |                 |
--|               |    Sort         |
--|               |_________________|
--|               |                 |
--|
procedure Sort(A : in out A_Type)
   is
   begin
   --|
   --|
   --|
   for I in reverse 2 .. A'Last loop
      --|
      --|   D E B U G
      --|   ---------
      Ada.Text_Io.Put_Line("Sort with i ="
                           & Integer'Image(I));
      --|
      --|  swap top and bottom
      --|
      Swap(A => A,
           I => 1,
           J => I);
      --|
      --|  forget about the last one
      --|
      Heap_Size := Heap_Size - 1;
      --|
      --|  move the new top in place in the heap
      --|
      Move_In_Heap(A => A,
                   I => 1);
   end loop;

end Sort;

begin
--|
--|
--|
Build_The_Heap(A => A);
--|
--|   D E B U G
--|   ---------
Ada.Text_Io.Put_Line("Heap built");
--|
--|
--|
Sort(A => A);
--|
--|  result to screen
--|
To_Screen(A => A);

end Exo_5;
```

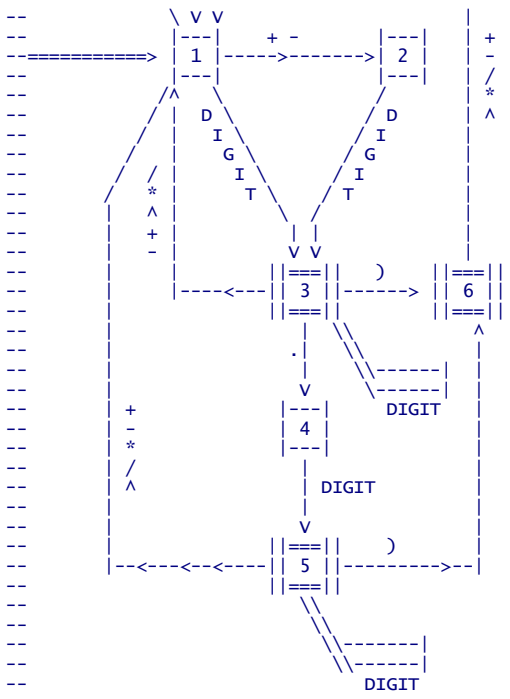
14.13 LE CODE DU TRI FUSION

```
--|                        M E R G E   S O R T

with Ada.Text_Io;
--|
--|               |_________________|
--|               |                 |
--|               |     Exo_6       |
--|               |_________________|
--|               |                 |
--|
procedure Exo_6
  is
  --| data and index types
  subtype Data_Type is Integer  range 1 .. 100;
  subtype Index_Type is Integer range 1 .. 7;
  --| array type
  type The_Array_Type is array (Index_Type range <>) of Data_Type;
  --|
  --|               |_________________|
  --|               |                 |
  --|               | To_Screen       |
  --|               |_________________|
  --|               |                 |
  --|
  procedure To_Screen(The_Array : in     The_Array_Type)
     is
     begin

     for I in The_Array'Range loop
        Ada.Text_Io.Put("|i="
                        & Integer'Image(I)
                        & " A="
                        & Integer'Image(The_Array(I)));
     end loop;

     Ada.Text_Io.New_Line;

  end To_Screen;
  --|
  --|               |_________________|
  --|               |                 |
  --|               |     Merge       |
  --|               |_________________|
  --|               |                 |
  --|
  procedure Merge(Array_To_Sort   : in out The_Array_Type;
                  Temporary_Array : in out The_Array_Type;
                  Low             : in     Index_Type;
                  Middle          : in     Index_Type;
                  High            : in     Index_Type)
     is
     I : Integer := Low;
     J : Integer := 1 + Middle;
     K : Integer := Low;
     begin
     --|
     --|  copy both halves into temporary
     --|
     Temporary_Array(Low .. High) := Array_To_Sort(Low .. High);
     --|
     --| merge the two sorted lists ( from both recursive calls ) :
     --|   Array_To_Sort(Low       .. middle)     and
     --|   Array_To_Sort(middle +1 .. High)  into one
     --|   Array_To_Sort(low       .. high)
     --|
     while I <= Middle and then J <= High loop
        --|
        --| store lowest one of Temporary_Array(i) or Temporary_Array(J)
        --| in Array_To_Sort(K)
        --| and increment index of the lowest side ( i or J)
        --| increment k (storage index)
        --|
        if Temporary_Array(I) <= Temporary_Array(J)
           then
           Array_To_Sort(K) := Temporary_Array(I);
           I                := I + 1;
           else
           Array_To_Sort(K) := Temporary_Array(J);
           J                := J + 1;
        end if;
        --|
        --|  increment storage index
        --|
        K := K + 1;
     end loop;
     --|
     --|  fill in data not used if any
     --|
     while I <= Middle loop
        Array_To_Sort(K) := Temporary_Array(I);
        K                := K + 1;
        I                := I + 1;
```

```
      end loop;
   end Merge;
   --|
   --|               |_________________|
   --|               |                 |
   --|               | Merge_Sort      |
   --|               |_________________|
   --|               |                 |
   --|
   procedure Merge_Sort(Array_To_Sort   : in out The_Array_Type;
                        Temporary_Array : in out The_Array_Type;
                        I               : in     Index_Type;
                        J               : in     Index_Type)
      is
      Middle : constant Index_Type := (I + J) / 2;
      begin
      Ada.Text_Io.Put_Line("Merge_Sort"
                           & Integer'Image(I)
                           & Integer'Image(J));
      --|
      --|   recursion ends ??
      --|
      if I >= J
         then
         return;
      end if;
      --|
      --|  recursive call on lower half
      --|
      Merge_Sort(Array_To_Sort   => Array_To_Sort,
                 Temporary_Array => Temporary_Array,
                 I               => I,
                 J               => Middle);
      --|
      --| recursive call on higher half
      --|
      Merge_Sort(Array_To_Sort   => Array_To_Sort,
                 Temporary_Array => Temporary_Array,
                 I               => Middle + 1,
                 J               => J);
      --|
      --|   Merge the two sorted lists ( from both recursive calls ) :
      --|   Array_To_Sort(Low       .. middle)     and
      --|   Array_To_Sort(middle +1 .. High)  into one
      --|   Array_To_Sort(low       .. high)
      --|
      Ada.Text_Io.Put_Line("merge" & Integer'Image(I) & Integer'Image(J));
      Merge(Array_To_Sort   => Array_To_Sort,
            Temporary_Array => Temporary_Array,
            Low             => I,
            Middle          => Middle,
            High            => J);

   end Merge_Sort;
   --|
   --| data entry (sort of)
   --|
   The_Array : The_Array_Type(Index_Type'Range) := (7, 3, 5, 2, 4, 1, 6);
   Dummy     : The_Array_Type(Index_Type'Range) := (others => Data_Type'Last);
   begin
   --|
   --|   call the sorting algo
   --|
   Merge_Sort(Array_To_Sort   => The_Array,
              Temporary_Array => Dummy,
              I               => The_Array'First,
              J               => The_Array'Last);
   --|
   --|  result to screen
   --|
   To_Screen(The_Array => The_Array);

end Exo_6;
```

14.14 LE CODE DU TEST DE L'EXPRESSION ALGÉBRIQUE: AUTOMATE

```
with Ada.Text_Io;
procedure Test_Arithmetic
   is

   --            (
   --          \----|
   --           \   | |---------<------------|
   --            \  | |                      |
```



```
--
-- 3, 5 and 6 are final states
--

type Transition_Type is
(Digit, Left_Parenthesis, Right_Parenthesis, Dot, Plus, Minus, Divide, Multiply, Power, Unknown);

type State_Type is new Integer range 0 .. 6;
type Matrix_Type is array (Transition_Type'Range, State_Type'Range) of State_Type;
type Look_Up_Table_Type is array (Character'Range) of Transition_Type;

Look_Up_Table : constant Look_Up_Table_Type := ('0' .. '9'=> Digit,
                                                '('       => Left_Parenthesis,
                                                ')'       => Right_Parenthesis,
                                                '.'       => Dot,
                                                '+'       => Plus,
                                                '-'       => Minus,
                                                '/'       => Divide,
                                                '*'       => Multiply,
                                                '^'       => Power,
                                               others     => Unknown);

--                                  transition     états  0  1  2  3  4  5  6

Matrix : constant Matrix_Type := (Digit            => (0, 3, 3, 3, 5, 5, 0),
                                  Left_Parenthesis  => (0, 1, 0, 0, 0, 0, 0),
                                  Right_Parenthesis => (0, 0, 0, 6, 0, 6, 0),
                                  Dot               => (0, 0, 0, 4, 0, 0, 0),
                                  Plus              => (0, 2, 0, 1, 0, 1, 1),
                                  Minus             => (0, 2, 0, 1, 0, 1, 1),
                                  Divide            => (0, 0, 0, 1, 0, 1, 1),
                                  Multiply          => (0, 0, 0, 1, 0, 1, 1),
                                  Power             => (0, 0, 0, 1, 0, 1, 1),
                                  Unknown           => (0, 0, 0, 0, 0, 0, 1));

State             : State_Type      := 1;
Transition        : Transition_Type := Transition_Type'First;
Current_Character : Character       := ' ';
User_S_Entry      : String(1 .. 50) := (others => ' ');
Last              : Integer         := 0;

begin
--|
--| fake user entry
--|
Last                    := 29;
User_S_Entry(1 .. Last) := "2+3/54.8*(4-7)^3.543*-5*(5-4)";
--|
--| run the finite states automaton
--|
for I in 1 .. Last loop
   Current_Character := User_S_Entry(I);
   Transition        := Look_Up_Table(Current_Character);
   State             := Matrix(Transition, State);

   Ada.Text_Io.Put_Line(Current_Character
                        & " "
                        & Transition_Type'Image(Transition)
                        & State_Type'Image(State));

end loop;
--|
```

```
   --| only 3, 5 and 6 ok
   --|
   Ada.Text_Io.Put(State_Type'Image(State));
end Test_Arithmetic;
```

14.15 LE CODE DU TEST DE L'EXPRESSION ALGÉBRIQUE: FONCTIONS

```
--
--|            UNIVERSITY TEACHING CLASS 2005 2006
--|               'ALGORITHMS USING ADVANCED ADA'
--|
--|Daniel@dgaudry.com
--|
--|  version 1.0   29th september 2005

with Ada.Text_Io;
--|
--|                |_________________|
--|                |                 |
--|                |   exo_3         |
--|                |_________________|
--|                |                 |
--|
procedure Exo_3
   is
   --|
   --|                        type definitions
   --|
   subtype User_Entry_Type is String(1 .. 100);

   type Data_Type is
      record
      Chaine                        : User_Entry_Type := (others => ' ');
      Result                        : Boolean         := True;
      Position_Of_Current_Character : Integer         := 1;
      Position_Of_Last_Character    : Integer         := 0;
   end record;
   --|
   --|                     global data
   --|
   Data : Data_Type;
   --|
   --|                |_________________________|
   --|                |                         |
   --|                |  Reading_Over           |
   --|                |_________________________|
   --|                |                         |
   --|
   function Reading_Over
      return Boolean
      is
      begin
      --|
      --|   return the result
      --|
      return Data.Position_Of_Last_Character + 1 = Data.Position_Of_Current_Character;
   end Reading_Over;
   --|
   --|                |_____________________________________|
   --|                |                                     |
   --|                |        Next                         |
   --|                |_____________________________________|
   --|                |                                     |
   --|
   procedure Next
      is
      begin
      --|
      --|   go to next index
      --|
      Data.Position_Of_Current_Character := 1 + Data.Position_Of_Current_Character;
   end Next;
   --|
   --|                |____________|
   --|                |            |
   --|                |  Get_Data  |
   --|                |____________|
   --|                |            |
   --|
   procedure Get_Data(Header : in     String;
                      Data   : in out Data_Type)
      is
      begin
      --|
      --|  put header on screen
      --|
      Ada.Text_Io.Put(Header & ' ');
      --|
      --|  get characters from user'input
      --|
      Ada.Text_Io.Get_Line(Data.Chaine, Data.Position_Of_Last_Character);
      --|
```

```
   --|   something's wrong!! capture error and return 0 and a string filled with spaces
   --|
   exception
   when others =>

   Data.Chaine                     := (others => ' ');
   Data.Position_Of_Last_Character := 0;
end Get_Data;
--|
--|               |_____________________|
--|               |                     |
--|               |  Current_Character  |
--|               |_____________________|
--|               |                     |
--|
function Current_Character
   return Character
   is
   begin
   --|
   --|   return the current character
   --|
   return Data.Chaine(Data.Position_Of_Current_Character);
end Current_Character;
--|
--|   declare them before hand to avoid problems with recursion order
--|
function Expression return Boolean;
function Term return Boolean;
function Factor return Boolean;
function Adding_Operator return Boolean;
function Multiplying_Operator return Boolean;
function Number return Boolean;
function Digit return Boolean;
--|
--|               |_____________________________|
--|               |                             |
--|               |        Expression           |
--|               |_____________________________|
--|               |                             |
--|
function Expression return Boolean
   is
   Dummy : Boolean;
   begin
   --EXPRESSION ::= [ ADDING_OPERATOR ] TERM { ADDING_OPERATOR TERM}

   Dummy := Adding_Operator;

   if not Term
      then
      return False;
   end if;

   while Adding_Operator loop

      if not Term
         then
         return False;
      end if;

   end loop;

   return True;
end Expression;
--|
--|               |_____________________________|
--|               |                             |
--|               |               Term          |
--|               |_____________________________|
--|               |                             |
--|
function Term return Boolean
   is
   Dummy : Boolean;
   begin
   --TERM ::= FACTOR { MULTIPLYING_OPERATOR [ADDING_OPERATOR] FACTOR}

   if not Factor
      then
      return False;
   end if;

   while Multiplying_Operator loop

      Dummy := Adding_Operator;

      if not Factor
         then
         return False;
      end if;
   end loop;

   return True;
end Term;
--|
--|               |_____________________________|
```

```
--|               |                                |
--|               |            Factor              |
--|               |________________________________|
--|               |                                |
--|
function Factor return Boolean
   is
   begin
   --FACTOR ::= NUMBER |  '('  EXPRESSION  ')'
   if Number
      then
      return True;

      elsif Current_Character = '('
      then
      Next;

      if not Expression
         then
         return False;
      end if;

      if Current_Character = ')'
         then
         Next;
         return True;
      end if;
   end if;

   return False;
end Factor;
--|
--|               |________________________________|
--|               |                                |
--|               |     Adding_Operator            |
--|               |________________________________|
--|               |                                |
--|
function Adding_Operator return Boolean
   is
   begin
   --ADDING_OPERATOR ::= '+'   |   '-'
   if Current_Character = '+'
      or else Current_Character = '-'
      then
      Next;
      return True;
   end if;

   return False;
end Adding_Operator;
--|
--|               |________________________________|
--|               |                                |
--|               |       Multiplying_Operator     |
--|               |________________________________|
--|               |                                |
--|
function Multiplying_Operator return Boolean
   is
   begin
   --MULTIPLYING_OPERATOR    ::= '*'   |  '/'   |   '^'
   if Current_Character = '*'
      or else Current_Character = '/'
      or else Current_Character = '^'
      then
      Next;
      return True;
   end if;

   return False;
end Multiplying_Operator;
--|
--|               |________________________________|
--|               |                                |
--|               |             Number             |
--|               |________________________________|
--|               |                                |
--|
function Number return Boolean
   is
   begin
   --NUMBER ::= DIGIT { DIGIT } [  '.'   DIGIT { DIGIT } ]
   if not Digit
      then
      return False;
   end if;

   while Digit loop
      null;
   end loop;

   if Current_Character = '.'
      then
      Next;

      if not Digit
```

```
            then
            return False;
         end if;

         while Digit loop
            null;
         end loop;

      end if;

      return True;
   end Number;
   --|
   --|               |_____________________________|
   --|               |                             |
   --|               |              Digit          |
   --|               |_____________________________|
   --|               |                             |
   --|
   function Digit return Boolean
      is
      begin
      --DIGIT ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

      case Current_Character
         is

         when '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'=>
         Next;
         return True;

         when others =>
         return False;

      end case;

   end Digit;
   --|
   --|               |_____________________________|
   --|               |                             |
   --|               |        To_Screen            |
   --|               |_____________________________|
   --|               |                             |
   --|
   procedure To_Screen
      is
      begin
      Ada.Text_Io.Put(Data.Chaine(1 .. Data.Position_Of_Last_Character)
                      & " 'validity is "
                      & Boolean'Image(Data.Result));

   end To_Screen;
   --|
   --|    M A I N
   --|    =======
   begin
   --|
   --|   data input
   --|
   Get_Data(Header => " expression ?? ",
            Data   => Data);
   --|
   --|   test validity
   --|
   Data.Result := Expression;
   --|
   --|   add check for characters unaccounted for at the end
   --|
   Data.Result := Data.Result and Reading_Over;
   --|
   --| put result on screen
   --|
   To_Screen;

end Exo_3;
```

14.16 LE CODE DE L'ARBRE DICTIONNAIRE

Le fichier poly_dic_tree.ads:

```
package Poly_Dic_Tree
is
--       COMMENTS will be appreciated
--       Daniel.Gaudry@wanadoo.fr


No_Such_A_Word : exception;
Wrong_Tag_Type : exception;
```

```
------------------------------------------------------------
--|            CHARACTER IN DICTIONARY TREE TYPE DECLARATION  |
--|                                                           |
--|  T O   B E   M O D I F I E D   A C C O R D I N G          |
--|          T O   Y O U R   N E E D S                        |
--|     currently set from space to ~                         |
------------------------------------------------------------|

subtype Character_Value_Type is Character range Character'Val(16#32#) .. Character'Val(16#7f#);
--|
--|             TAGGED TYPE DECLARATION
--|
type Tagged_Data_Type is abstract tagged null record;
type Tagged_Data_Access_Type is access all Tagged_Data_Type'Class;
--|
--|             NODE TYPE DECLARATION
--|
type Dictionary_Tree_Type is limited private;
--|
--|              ||
--|              | Clear_Dictionary  |
--|              |___________________|
--|
procedure Clear_Dictionary(The_Dictionary_Tree : in out Dictionary_Tree_Type);
--|
--|              ||
--|              | Add_To_Dictionary |
--|              |___________________|
--|
procedure Add_To_Dictionary(The_Record            : in     Tagged_Data_Type'Class;
                            In_The_Dictionary_Tree : in out Dictionary_Tree_Type;
                            With_The_Label         : in     String);
--|
--|              ||
--|              | Print_Dictionary |
--|              |__________________|
--|
procedure Print_Dictionary(Dictionary_Tree : in     Dictionary_Tree_Type);
--|
--|              ||
--|              | Stat |
--|              |______|
--|
procedure Stat(How_Many_Nodes_Created :    out Long_Integer);
--|
--|              ||
--|              | Get_Record_Access_From_Label |
--|              |______________________________|
--|
procedure Get_Record_Access_From_Label(From_The_Dictionary_Tree : in out Dictionary_Tree_Type;
                                       And_The_Label            : in     String;
                                       Tagged_Data_Access       :    out Tagged_Data_Access_Type);

-----------------------------------------------
---      P A S S I V E   I T E R A T O R     ---
-----------------------------------------------

generic
--|
--|              ||
--|              | Iterate |
--|              |_________|
--|
with procedure Iterate(The_Node_Access : in out Tagged_Data_Access_Type;
                       Next_One        : in out Boolean;
                       The_Label       : in out String);
--|
--|              ||
--|              | Visit_All |
--|              |___________|
--|
procedure Visit_All(Dictionary_Tree : in     Dictionary_Tree_Type);

-----------------------------------------------
-----------------------------------------------
-----------------------------------------------

private

type Node_Access_Type is access Dictionary_Tree_Type;

type Node_Access_Array_Type is array (Character_Value_Type'Range) of Node_Access_Type;

type Dictionary_Tree_Type is
record
   End_Of_Word        : Boolean                := False;
   Node_Access_Array  : Node_Access_Array_Type  := (others => null);
   Character_Value    : Character_Value_Type    := Character_Value_Type'First;
   Tagged_Data_Access : Tagged_Data_Access_Type := null;
end record;

Local_Hash_Value : Integer      := 0;
Index            : Integer      := 0;
Node_Num         : Long_Integer := 0;

end Poly_Dic_Tree;
```

Le fichier poly_dic_tree.adb:

```
with Ada.Text_Io;
with Ada.Tags;

package body Poly_Dic_Tree
is

--|
--|              ||
--|              | Clear_Dictionary  |
--|              |___________________|
--|
   procedure Clear_Dictionary(The_Dictionary_Tree : in out Dictionary_Tree_Type)
   is
   --|
   --|              ||
   --|              | Clear_Next_Letter|
   --|              |__________________|
   --|
      procedure Clear_Next_Letter(Dictionary_Tree : in out Dictionary_Tree_Type)
      is
      begin
      --|
      --| loop On all Possible Char at That Level
      --|
      for Char in Character_Value_Type'Range loop
         --|
         --|  is is the end  no contineue loop yes recursive call
         --|
         if Dictionary_Tree.Node_Access_Array(Char) /= null
            then
            --|
            --| recursive call
            --|
            Clear_Next_Letter(Dictionary_Tree => Dictionary_Tree.Node_Access_Array(Char).all);
            --|
            --| back from recursive call clear node
            --|
            Dictionary_Tree.Node_Access_Array(Char) := null;
            Dictionary_Tree.Tagged_Data_Access      := null;
         end if;
      end loop;
      end Clear_Next_Letter;
   --|
   --|              ||
   --|              |main  Clear_Dictionary |
   --|              |_____________________|
   --|
   begin
   --|
   --|start customary recursion
   --|
   Clear_Next_Letter(Dictionary_Tree => The_Dictionary_Tree);

   Index    := 0;
   Node_Num := 0;

   end Clear_Dictionary;
--|
--|              ||
--|              | Add_To_Dictionary |
--|              |___________________|
--|
   procedure Add_To_Dictionary(The_Record             : in     Tagged_Data_Type'Class;
                               In_The_Dictionary_Tree : in out Dictionary_Tree_Type;
                               With_The_Label         : in     String)
   is
   --|
   --|              ||
   --|              | Add_Word |
   --|              |__________|
   --|
      procedure Add_Word(In_The_Dictionary_Tree : in out Dictionary_Tree_Type;
                         With_The_Label         : in     String)
      is
      Char : Character := With_The_Label(With_The_Label'First);
      begin
      --|
      --| SUCH A CHARACTER ALREADY EXISTS IN THE DICTIONNARY ?
      --|
      if In_The_Dictionary_Tree.Node_Access_Array(Char) = null
         then
         --|
         --| NO, CREATE THE NEW NODE AND ASSIGN CHARACTER VALUE
         --|
         Node_Num                                      := 1 + Node_Num;
         In_The_Dictionary_Tree.Node_Access_Array(Char) := new Dictionary_Tree_Type;
         --|
         --| MAKE IT BLANK
         --|
         In_The_Dictionary_Tree.Node_Access_Array(Char).End_Of_Word       := False;
         In_The_Dictionary_Tree.Node_Access_Array(Char).Node_Access_Array := (others => null);
         --|
         --| ASSIGN CHAR VALUE
```

```
         --|
         In_The_Dictionary_Tree.Node_Access_Array(Char).Character_Value := Char;
      end if;
      --|
      --| IS IT THE LAST CHARACTER
      --|
      if With_The_Label'First = With_The_Label'Last
         then
         --|
         --| IS IT ALSO THE END OF A WORD PREVIOUSLY RECORDED
         --|
         if In_The_Dictionary_Tree.Node_Access_Array(Char).End_Of_Word = True
            then
            null;
            else
            --|
            --| NO  RECORD IT AND UPDATE
            --|
            In_The_Dictionary_Tree.Node_Access_Array(Char).End_Of_Word := True;
         end if;
         --|
         --|  LAST LETTER OF WORD => LOAD/UPDATE CLASS DATA
         --|
         In_The_Dictionary_Tree.Node_Access_Array(Char).Tagged_Data_Access := new Tagged_Data_Type'Class'(The_Record);
         else
         --|
         --|REMOVE ONE CHAR FROM THE WORD AND CALL AGAIN WITH THIS NODE
         --|
         Add_Word(In_The_Dictionary_Tree => In_The_Dictionary_Tree.Node_Access_Array(Char).all,
                  With_The_Label         => With_The_Label(1 + With_The_Label'First .. With_The_Label'Last));
      end if;
      end Add_Word;
   --|
   --|              ||
   --|              | main Add_To_Dictionary |
   --|              |________________________|
   --|
   begin
   Add_Word(In_The_Dictionary_Tree => In_The_Dictionary_Tree,
            With_The_Label         => With_The_Label);
   end Add_To_Dictionary;
--|
--|              ||
--|              | Print_Dictionary |
--|              |__________________|
--|
   procedure Print_Dictionary(Dictionary_Tree : in     Dictionary_Tree_Type)
   is
   S                 : String(1 .. 200);
   Depth_Of_Recursion : Integer := 0;
   --|
   --|              ||
   --|              | Next_Letter |
   --|              |_____________|
   --|
      procedure Next_Letter(Dictionary_Tree : in     Dictionary_Tree_Type)
      is
      begin
      --|
      --| KEEP  TRACK OF RECURSION LEVEL
      --|
      Depth_Of_Recursion := 1 + Depth_Of_Recursion;
      --|
      --| LOOP ON ALL POSSIBLE CHAR AT THAT LEVEL
      --|
      for Char in Character_Value_Type'Range loop
         --|
         --| IF NULL NO SUCH CHAR WAS RECORDED
         --|
         if Dictionary_Tree.Node_Access_Array(Char) /= null
            then
            --|
            --|LOAD CHAR
            --|
            S(Depth_Of_Recursion) := Dictionary_Tree.Node_Access_Array(Char).Character_Value;
            --|
            --| IF END OF WORD PRINT
            --|
            if Dictionary_Tree.Node_Access_Array(Char).End_Of_Word
               then
               Ada.Text_Io.Put("->" & S(1 .. Depth_Of_Recursion) & Ascii.Lf);
            end if;
            --|
            --|RECURSE ONCE MORE
            --|
            Next_Letter(Dictionary_Tree => Dictionary_Tree.Node_Access_Array(Char).all);
         end if;
      end loop;
      --|
      --| UPDATE RECUSION LEVEL
      --|
      Depth_Of_Recursion := Depth_Of_Recursion - 1;
      end Next_Letter;
   --|
   --|              ||
   --|              | main Print_Dictionary |
   --|              |_______________________|
```

```
   --|
   begin
   --|
   --| CLEAN AND GO
   --|
   S := (S'Range               => ' ');
   Next_Letter(Dictionary_Tree => Dictionary_Tree);
   end Print_Dictionary;
--|
--|              ||
--|              | Stat |
--|              |______|
--|
   procedure Stat(How_Many_Nodes_Created :    out Long_Integer)
   is
   begin
   How_Many_Nodes_Created := Node_Num;
   end Stat;
--|
--|              ||
--|              | Visit_All |
--|              |___________|
--|
   procedure Visit_All(Dictionary_Tree : in     Dictionary_Tree_Type)
   is
   S                  : String(1 .. 200);
   Depth_Of_Recursion : Integer := 0;
   Next_One           : Boolean := True;
   --|
   --|              ||
   --|              | Next_Letter |
   --|              |_____________|
   --|
      procedure Next_Letter(Dictionary_Tree : in     Dictionary_Tree_Type)
      is
      begin
      if not Next_One
         then
         return;
      end if;
      --|
      --| KEEP  TRACK OF RECURSION LEVEL
      --|
      Depth_Of_Recursion := 1 + Depth_Of_Recursion;
      --|
      --| LOOP ON ALL POSSIBLE CHAR AT THAT LEVEL
      --|
      for Char in Character_Value_Type'Range loop
         --|
         --| IF NULL NO SUCH CHAR WAS RECORDED
         --|
         if Dictionary_Tree.Node_Access_Array(Char) /= null
            then
            --|
            --|LOAD CHAR
            --|
            S(Depth_Of_Recursion) := Dictionary_Tree.Node_Access_Array(Char).Character_Value;
            --|
            --| IF END OF WORD return to main
            --|
            if Dictionary_Tree.Node_Access_Array(Char).End_Of_Word
               then

               Iterate(The_Node_Access => Dictionary_Tree.Node_Access_Array(Char).Tagged_Data_Access,
                       Next_One        => Next_One,
                       The_Label       => S(1 .. Depth_Of_Recursion));
            end if;
            --|
            --|RECURSE ONCE MORE if user wants it
            --|
            if Next_One
               then
               Next_Letter(Dictionary_Tree => Dictionary_Tree.Node_Access_Array(Char).all);
            end if;
         end if;
      end loop;
      --|
      --| over ?
      --|
      if not Next_One then
         return;
      end if;
      --|
      --| UPDATE RECUSION LEVEL
      --|
      Depth_Of_Recursion := Depth_Of_Recursion - 1;
      end Next_Letter;
   --|
   --|              ||
   --|              |main  Visit_All |
   --|              |________________|
   --|
   begin
   --|
   --| CLEAN AND GO
   --|
   S := (S'Range               => ' ');
```

```
   Next_Letter(Dictionary_Tree => Dictionary_Tree);
   end Visit_All;
--|
--|              ||
--|              | Get_Record_Access_From_Label |
--|              |______________________________|
--|
   procedure Get_Record_Access_From_Label(From_The_Dictionary_Tree : in out Dictionary_Tree_Type;
                                          And_The_Label            : in     String;
                                          Tagged_Data_Access       :    out Tagged_Data_Access_Type)
   is
   --|
   --|              ||
   --|              | Seek_Word |
   --|              |___________|
   --|
      procedure Seek_Word(From_The_Dictionary_Tree : in out Dictionary_Tree_Type;
                          And_The_Label            : in     String)
      is
      Char : Character := And_The_Label(And_The_Label'First);
      begin
      --|
      --| SUCH A CHARACTER EXISTS IN THE DICTIONNARY ?
      --|
      if From_The_Dictionary_Tree.Node_Access_Array(Char) = null
         then
         raise No_Such_A_Word;
      end if;
      --|
      --| LAST LETTER
      --|
      if And_The_Label'First = And_The_Label'Last
         then
         --|
         --|IS IT ALSO THE END OF THE WORD
         --|
         if True = From_The_Dictionary_Tree.Node_Access_Array(Char).End_Of_Word
            then
            --|
            --| LOAD SEEKED FOR DATA
            --|
            Tagged_Data_Access := From_The_Dictionary_Tree.Node_Access_Array(Char).Tagged_Data_Access;
            else
            --|
            --| IF END OF WORD WITHOUT END OF WORD
            --|
            raise No_Such_A_Word;
         end if;
         else
         --|
         --|REMOVE ONE CHAR FROM THE WORD AND CALL AGAIN WITH THIS NODE
         --|
         Seek_Word(From_The_Dictionary_Tree => From_The_Dictionary_Tree.Node_Access_Array(Char).all,
                   And_The_Label            => And_The_Label(1 + And_The_Label'First .. And_The_Label'Last));
      end if;
      end Seek_Word;
   --|
   --|              ||
   --|              |main  Get_Record_Access_From_Label |
   --|              |___________________________________|
   --|
   begin
   --|
   --| START RECURSION
   --|
   Seek_Word(From_The_Dictionary_Tree => From_The_Dictionary_Tree,
             And_The_Label            => And_The_Label);
   end Get_Record_Access_From_Label;
-----------------------------------------------
-----------------------------------------------
-----------------------------------------------

end Poly_Dic_Tree;
```

15 L'ALGORITHMIQUE EN ACTION, EXEMPLE 1: UN CALCULATEUR SIMPLE

Nous allons réaliser un calculateur simple : les 4 opérations sans parenthèses (espaces non significatifs : 2+3 équivalent à 2 + 3).

15.1 ANALYSE

15.1.1 Première approche du découpage

La décomposition du problème est assez simple:

1. Éliminer les espaces
2. Identifier le début des opérateurs
3. identifier le début des opérandes
4. Découper en opérateurs et opérandes (début .. fin-1)
5. Traiter la fin de chaîne spécialement pour stoker l'opérande final.

exemple : 2/-3 sera découpé en

2	(opérande)
/	(opérateur)
-3	(opérande)

15.1.2 Entrée du calcul

Sous forme d'un chaîne de caractères par l'utilisateur:

- Soit comme paramètre d'appel : main 2+3/4
- Soit comme réponse à une question si pas de paramètre d'appel

15.1.3 Test de la validité

2+3 est valide 2+*3 n'est pas valide => rejet si non valide:
utilisation d'une grammaire et d'automates à états finis.

15.1.4 Traduction du découpage précédent en postfix pour le calcul

(élimination du problème de la priorité des opérateurs) 2+3 => 23+1, 5/3/2 => (15/3)/2 =2.5 et non pas 15/(3/2).

15.1.5 Calcul à partir de la traduction postfix.

La solution classique du stack sera employée.

15.1.6 Affichage du résultat

Traduction postfix et résultat du calcul.

15.1.7 Découpage en tâches indépendantes

Ces tâches sont intellectuellement indépendantes. Dans la réalité on peut imaginer plusieurs équipes chacune occupée à la solution de sa partie.

Le codage de chacune des des parties nous permettra d'apprendre le langage progressivement et de voir la construction d'une projet informatique.

15.1.8 Dépendances des modules

1. Une lecture attentive montre que les parties 1, 2 et 3 **utilisent** la chaîne de caractères entrée par l'utilisateur.
2. Les parties 2 et 3 **nécessitent la lecture de la chaîne** de caractères.
3. La partie 4 utilise le résultat de la partie 3.
4. La partie 5 utilise le résultat de la partie 4.

il nous faudra définir en détails les **modes** et **format** des **données** pour la transmission entre 3 => 4 => 5.

De même, il faudra i**dentifier les opérations sur la chaîne de caractères communes à 1, 2 et 3** . Elles seront implémentées séparément dans la partie 6
qui sera donc utilisée par 1, 2 et 3 dans l'interaction avec la chaîne de caractères entrée par l'utilisateur.

15.2 DÉFINITIONS DES DONNÉES

15.2.1 Identification des données

Les données étant utilisées par plusieurs parties (packages) différents, nous centraliseront les types de ces données dans un package unique, utilisé par tous les autres packages.
Ce package sera appelé data.ads NOTER L'EXTENTION **ADS**
Nous devrons manipuler par exemple :
-2
*
5.31
/
3.018
c'est un tableau de chaînes de caractères de longueurs différentes:
2
1
4
1
5

15.2.2 Le stockage des données

Le stockage et la manipulation de tableau de chaîne**s** de caractères de longueurs différentes en ada impose un tableau de pointeurs vers des chaînes de caractères.

15.2.3 Les types de données

Il se défini comme suit:

15.2.3.1 Chaîne de caractères

le pointeur vers une chaîne de caractères:

type string_access_type is access all string;

15.2.3.2 Tableau de chaînes de caractères

le tableau de chaînes de caractères:

type string_access_table_type is array (integer range <>) of string_access_type;

l'entrée utilisateur sera stockée dans une variable de type string_access_type;

la déclaration des variables sera effectuée au début du programme principal

15.2.4 Les données dans la spécification d'un « package » commun

15.2.4.1 Le code

Le code de data.ads peut s'écrire comme suit:

```
package data
is
```

Le pointeur vers une chaîne de caractères:

```
type string_access_type is access all string;
```

le tableau de chaînes de caractères

```
type string_access_table_type is array (integer range <>) of string_access_type;

end data;
```

L'utilisation (programme principal) pourrait être :
with data;
.....
déclaration et initialisation

```
tableau: string_access_table_type(1..10):=(others=> null);

user_input: string_access_type:=null;
```

stockage

```
tableau(i):= new string'(my_string);
```

lecture (dé-référencement)

```
x....:=tableau(i).all;
```

15.3 RELATIONS ENTRE LES MODULES

Le module définition des données

15.3.0.1 Nom du module

Data.ads

15.3.0.2 Données en sortie

Les type de données servant au stockage de l'entrée utilisateur et au traitement.

15.3.1 Le module principal

15.3.1.1 Nom du module

Main.adb

15.3.1.2 Données à l'entrée

Les déclarations de variables:

Un pointeur sur une chaîne de caractères pour la sortie de l'entrée des données et l'entrée découpage.

Une matrice de pointeurs sur des chaînes de caractères pour la sortie du découpage et l'entrée de la traduction postifx.

Une autre matrice de pointeurs sur des chaînes de caractères pour la sortie de la traduction postfix et l'entrée calcul.

Un réel pour la sortie calcul et l'entrée affichage.

15.3.1.3 Traitement des données

LE MODULE PRINCIPAL NE FAIT AUCUN TRAITEMENT. Il fait la liaison entre les autres modules. Il peut toutefois contenir boucles et tests pour faire plusieurs fois le traitement.

Il fait les appel successifs aux autres modules.
Entry
Validation
decoupage
postfix
calcul
affichage

15.3.1.4 Données en sortie

Aucune.

15.3.2 Le module entrée utilisateur

15.3.2.1 Nom du module

Utilisateur en deux fichiers : data_io.ads et data_io.adb

15.3.2.2 Données à l'entrée

Aucune.

15.3.2.3 Traitement des données

S'il existe des paramètres d'appel, il retourne ces paramètres, sinon il demande à l'utilisateur le calcul à effectuer.

15.3.2.4 Données en sortie

Le calcul à effectuer en un pointeur sur une chaîne de caractères.

15.3.3 Le module test de la validité

15.3.3.1 Nom du module

Deux fichiers Validation.ads etValidation.adb

15.3.3.2 Données à l'entrée

Le calcul à effectuer (un pointeur sur une chaîne de caractères).

15.3.3.3 Traitement des données

Test de grammaire

15.3.3.4 Données en sortie

Booléen vrai faux

15.3.3.5 Le Module découpage en opérateur opérande

15.3.3.5.1 Nom du module

Parse en deux fichiers decoupage.ads et decoupage.adb

15.3.3.5.2 Données à l'entrée

Le calcul (un pointeur sur une chaîne de caractères) sous une forme grammaticalement correcte. Cela simplifie beaucoup le code car on est sur qu'il n'y a pas d'erreurs de syntaxe.

15.3.3.5.3 Traitement des données

Découpage en opérateur et opérande.

15.3.3.5.4 Données en sortie

Une matrice de pointeurs sur des chaînes de caractères contenant opérateur et opérandes dans l'ordre prefix habituel. Sans espaces au début de la chaîne.

15.3.4 Le module traduction du découpage précédent en postfix

15.3.4.1 Nom du module

Traduction.adb et Traduction.ads

15.3.4.2 Données à l'entrée

Tableau dans l'ordre infix

15.3.4.3 Traitement des données

Traduction et postfix

15.3.4.4 Données en sortie

Tableau dans l'ordre postfix.

15.3.5 Le module Calcul à partir du postfix

15.3.5.1 Nom du module

Calcul.adb et Calcul.ads

15.3.5.2 Données à l'entrée

Tableau dans l'ordre postfix.

15.3.5.3 Traitement des données

Calcul avec un stack de réels.

15.3.5.4 Données en sortie

Le réel qui est le résultat du calcul.

15.3.6 Le module affichage du résultat

15.3.6.1 Nom du module

Dat_io.ads et Data_io. adb

15.3.6.2 Données à l'entrée

Le réel résultat du calcul

15.3.6.3 Traitement des données

Mise en forme avec format xxx.xxxx sans espaces ni zeros non significatifs
Impression écran.

15.3.6.4 Données en sortie

Acune

15.3.7 Le code : le point de départ

Ce code, **NE FAISANT AUCUN TRAITEMENT DES DONNES,** peut maintenant être écrit en ce qui concerne:

- les modules
- les liens entre modules
- les appels des fonctions

- **Il compile**

- Il ne fait rien
- chaque module renvoie la valeur attendue pour l'exemple 2+3/4

Il faut maintenant remplir chaque fonction/ procédure dans chaque module ce qui permet une division du travail ou un travail successif sur chaque module.

Voici le code fichier par fichier

15.3.7.1 data.ads

```
package Data
  is

  --Le pointeur vers une chaîne de caractères:

  type String_Access_Type is access all String;

  --le tableau de chaînes de caractères
  subtype Table_Size is Integer range 1 .. 50;

  type String_Access_Array_Type
  is array(Integer range <>) of String_Access_Type;

  subtype String_Access_Table_Type
  is String_Access_Array_Type(Table_Size'Range);

end Data;
```

15.3.7.2 main.adb

```
-- calculateur simple avc traduction infix postfix
with Data;
```

```
  with Validation;
  with Decoupage;
  with Data_Io;
  with Calcul;
  with Traduction;

  procedure Main

    is

    Infix   : Data.String_Access_Table_Type := (others => null);
    Postfix : Data.String_Access_Table_Type := (others => null);
    Input   : Data.String_Access_Type       := null;
    Result  : Float                         := 0.0;

    begin

    Input := Data_Io.Data_Entry;

    if Validation.Valid_User_Input(User_Input => Input)
       then

       Infix := Decoupage.Parse(User_Input => Input);

       Postfix := Traduction.Infix_To_Postfix(From => Infix);

       Result := Calcul.Calcul_From_Postfix(Postfix => Postfix);

       Data_Io.To_Screen(Value => Result);

    end if;
  end Main;
```

15.3.7.3 calcul.ads

```
  with Data;
  package Calcul
    is
    function Calcul_From_Postfix(Postfix : in     Data.String_Access_Table_Type)
    return Float;
  end Calcul;
```

15.3.7.4 calcul.adb

```
package body Calcul
    is
    function Calcul_From_Postfix(Postfix : in     Data.String_Access_Table_Type)
       return Float
       is
       begin
       return 2.0 + 3.0 / 4.0;
    end Calcul_From_Postfix;
end Calcul;
```

15.3.7.5 data_io.ads

```
 with Data;

  package Data_Io
    is

    function Data_Entry return Data.String_Access_Type;

    procedure To_Screen(Value : in     Float);

  end Data_Io;
```

15.3.7.6 data_io.adb

```
 with Data;
  with Ada.Text_Io;


  package body Data_Io
    is

    function Data_Entry
       return Data.String_Access_Type
       is
       begin
       return new String'("2+3/4");

    end Data_Entry;

    procedure To_Screen(Value : in     Float)
       is
```

```
      begin
      Ada.Text_Io.Put(Float'Image(Value));
   end To_Screen;
end Data_Io;
```

15.3.7.7 decoupage.ads

```
with Data;
package Decoupage
   is


   function Parse(User_Input : in     Data.String_Access_Type)
   return Data.String_Access_Table_Type;
end Decoupage;
```

15.3.7.8 decoupage.adb

```
package body Decoupage
   is


   function Parse(User_Input : in     Data.String_Access_Type)
      return Data.String_Access_Table_Type
      is
      Dummy : Data.String_Access_Table_Type;
      begin
      Dummy(1) := new String'("2");
      Dummy(2) := new String'("+");
      Dummy(3) := new String'("3");
      Dummy(4) := new String'("/");
      Dummy(5) := new String'("4");
      return Dummy;
   end Parse;
end Decoupage;
```

15.3.7.9 traduction.ads

```
with Data;

package Traduction
   is
   function Infix_To_Postfix(From : in     Data.String_Access_Table_Type)
   return Data.String_Access_Table_Type;
end Traduction;
```

15.3.7.10 traduction.adb

```
with Data;

package body Traduction
   is

   function Infix_To_Postfix(From : in     Data.String_Access_Table_Type)
      return Data.String_Access_Table_Type
      is
      Dummy : Data.String_Access_Table_Type;
      begin
      Dummy(1) := new String'("2");
      Dummy(2) := new String'("3");
      Dummy(3) := new String'("4");
      Dummy(4) := new String'("/");
      Dummy(5) := new String'("+");
      return Dummy;
   end Infix_To_Postfix;

end Traduction;
```

15.3.7.11 validation.ads

```
with Data;

package Validation
   is
   function Valid_User_Input(User_Input : in     Data.String_Access_Type)
   return Boolean;

end Validation;
```

15.3.7.12 validation.adb

```
with Data;

package body Validation
   is
   function Valid_User_Input(User_Input : in     Data.String_Access_Type)
      return Boolean
      is
      begin
      return True;
   end Valid_User_Input;

end Validation;
```

15.4 LE TRAITEMENT DES DONNÉES

Maintenant, nous allons remplir le code des modules les uns après les autres.

15.4.1 Le module définition des données

15.4.1.1 Définition des procédure et fonctions

Aucune procédure ni fonctions déjà écrit dans la phase préliminaire précédente.

15.4.1.2 Traitement

Aucun .

15.4.2 Le module principal

15.4.2.1 Définition des procédure et fonctions

C'est la procédure principale qui appelle toutes les autres

15.4.2.2 Traitement

```
With ................;

Procedure main
is
infix: data.string_access_type:=(others=>null);
postfix: data.string_access_type:=(others=>null);
input: string_access_type:=null;
result: float;
begin

input:=data_Entry;

if Validation.valid_user_input(user_input=>input)
then

infix:=decoupage.parse (user_input=>input) ;

postfix:=traduction.infix_to_postfix ( from =>infix);

result:=calcul.calcul_from_postfix (postfix =>postfix);

affichage.to_screen(resultat => result);

end if;
end main:
```

15.4.3 Le module entrée utilisateur

15.4.3.1 Définition des procédure et fonctions

```
 function Data_Entry
       return Data.String_Access_Type
```

15.4.3.2 Traitement

Lit la ligne de commande, par exemple ./main 2+3 -9/8
si elle est absente demande à l'utilisateur.

15.4.3.2.1 Lecture de la commande

C'est très spécifique à un langage .

15.4.3.2.2 Analyse du problème

Deux choix :les paramètres sont présents dans la ligne de commande ou entrée après demande si pas de paramètre.

15.4.3.2.3 Lecture de la ligne de commande

A priori, je ne connais pas le comportement du package command line. je sais en lisant la doc que:

1. j'ai accès au nombre d'arguments :

   ```
   Ada.Command_Line.Argument_Count
   ```

2. je peux obtenir tous les arguments en utilisant le numéro de cet argument:

   ```
   Ada.Command_Line.Argument(Number => k)
   ```

3. si aucun argument
   `Ada.Command_Line.Argument_Count`
   renvoie zéro et cela permet **un test facile pour savoir si il y a des arguments ou si il faut demander à l'utilisateur.**

Le fait qu'il peut y avoir **un nombre variable d'arguments** de 1 à
`Ada.Command_Line.Argument_Count`
et qu'il faut **tous successivement les traiter** indique **une boucle for** :

Par exemple :

```
for k in 1 .. Ada.Command_Line.Argument_Count loop
     .....
end loop;
```

nous utiliserons k comme numéro courant de l'argument k ira de 1 à
`Ada.Command_Line.Argument_Count`

pour en savoir plus il faut écrire un morceau de code qui affichera tous les arguments dans l'ordre imposé par l'os employé.

```
for k in 1 .. Ada.Command_Line.Argument_Count loop

      ada.text_io.put_line(" cmd line "
      & Integer'Image(k)
      & " >"
      & Ada.Command_Line.Argument(Number => k)
      & '<'
      );

 end loop;
```

les " >" et '<' servent à vérifier qu'il n'y a pas d'espace avant ou après l'argument, ce qui modifierait le code. On affichera aussi le numéro courant.
Si > z< il y a un espace avant qui ne serait pas détecté sans cette précaution.

Après compilation procédons à des essais :

Essai 1 pas d'espaces dans la ligne de commande :

```
[daniel@localhost simple_calc_solution]$ ./main 1+2
 cmd line  1 >1+2<
 2.75000E+00
```

On observe un seul argument (la boucle s'arrête après 1) et aucun espace ni avant ni après.

```
[daniel@localhost simple_calc_solution]$ ./main 1 + 2
 cmd line  1 >1<
 cmd line  2 >+<
 cmd line  3 >2<
 2.75000E+00
```

On observe 3 arguments (la boucle s'arrête après 3) et aucun espace ni avant ni après. **Les espaces dans la ligne de commande sont interprétés par l'os comme des séparateurs**.

```
[daniel@localhost simple_calc_solution]$ ./main 1 + 2 /33.2
 cmd line  1 >1<
 cmd line  2 >+<
 cmd line  3 >2<
 cmd line  4 >/33.2<
 2.75000E+00
```

Les essai suivants confirment : aucun espace ni avant ni après. Les espaces dans la ligne de commande sont interprétés par l'O.S. comme des séparateurs.

```
[daniel@localhost simple_calc_solution]$ ./main 1 + 2 / 33.2
 cmd line  1 >1<
 cmd line  2 >+<
 cmd line  3 >2<
 cmd line  4 >/<
 cmd line  5 >33.2<
 2.75000E+00
[daniel@localhost simple_calc_solution]$ ./main 1 + 2 / -33.2
 cmd line  1 >1<
 cmd line  2 >+<
 cmd line  3 >2<
 cmd line  4 >/<
 cmd line  5 >-33.2<
 2.75000E+00
[daniel@localhost simple_calc_solution]$ ./main 1 + 2 / - 33.2
 cmd line  1 >1<
 cmd line  2 >+<
 cmd line  3 >2<
 cmd line  4 >/<
 cmd line  5 >-<
 cmd line  6 >33.2<
 2.75000E+00
```

Il nous faut donc construire la réponse utilisateur à partir de la ligne de commande en mettant bout à bout les différentes parties renvoyées par l'O.S..
Nous déclarerons une chaîne de caractères (elle servira aussi pour la demande si il n'y a aucun argument dans la ligne de commande).

Chaîne : string (1..50):=(others=>' ');

Une chaîne de 50 caractères initialisée avec des espaces dans toutes les positions de 1 à 50.
le (others signifie toutes les positions et => ' ') la valeur à leur donner.
Il nous faut maintenant décomposer le problème de l'ajout successif de tous les arguments:
le type string étant un tableau, le langage oblige à une compatibilité des positions.
Chaîne := Ada.Command_Line.Argument(Number => k)
ne marchera pas car chaîne a une longueur de 1 .. 50 et la **longueur de Ada.Command_Line.Argument(Number => k) est inconnue et variable!!!!!!!!**
il faut donc garder la première position libre dans chaîne (1 au début) dans une variable (par exemple position).
Il faut aussi pouvoir connaître la longueur de l'argument.

- Déclaration:

```
command_line_length : integer :=0;
```

- Usage:

```
command_line_length :=Ada.Command_Line.Argument(Number => k)'length;
```

nous pouvons maintenant insérer à chaque tour de la boucle les arguments un par un

Insertion attention au problème des piquets et des intervalles !!!! le -1 est très important!!

```
chaîne(position..position+command_line_length-1):= Ada.Command_Line.Argument(Number => k);
```

Mise à jour de la position pour l'insertion suivante:

```
position := position+command_line_length;
```

A la fin de la boucle il faut remettre à jour la position car elle est prête pour l'insertion suivante c'est à dire **une position trop loin !!** la syntaxe originale peut permettre de souligner.

Position := - 1 + Position;

Mise en forme finale du code pour cette partie

```
.....
.....
       Command_Line_Length : Integer         := 0;
       Position            : Positive        := 1;
       Chaine              : String(1 .. 50) := (others => ' ');
```

```
      begin

      for K in 1 .. Ada.Command_Line.Argument_Count loop

         Command_Line_Length := Ada.Command_Line.Argument(Number => K)'Length;

         Chaine(Position .. Position + Command_Line_Length - 1)
         := Ada.Command_Line.Argument(Number => K);

         Position := Position + Command_Line_Length;

      end loop;

      Position := - 1 + Position;
      -- debug

      Ada.Text_Io.Put_Line('>'& Chaine(1 .. Position) & '<');
```

15.4.3.2.4 Entrée par l'utilisateur s'il n'y a pas d'arguments sur la ligne de commande.

La premiére chose à faire est de tester la longueur de la ligne de commande
si 0 demander à l'utilisateur. Sinon lire la ligne de commande
Le test est très simple:

```
.............
 if 0 = Ada.Command_Line.Argument_Count
         then
.........
```

Affichage de la demande :

```
         Ada.Text_Io.Put(" enter calculation : ");
```

Lecture de l'entrée clavier:

```
         Ada.Text_Io.Get_Line(Chaine, Position);
```

15.4.3.3 Le code complet

```
   function Data_Entry
      return Data.String_Access_Type
      is
      Command_Line_Length : Integer         := 0;
      Position            : Positive        := 1;
      Chaine              : String(1 .. 50) := (others => ' ');
      begin

      if 0 = Ada.Command_Line.Argument_Count
         then

         Ada.Text_Io.Put(" enter calculation : ");
         Ada.Text_Io.Get_Line(Chaine, Position);

         else

         for K in 1 .. Ada.Command_Line.Argument_Count loop

            Command_Line_Length := Ada.Command_Line.Argument(Number => K)'Length;

            Chaine(Position .. Position + Command_Line_Length - 1)
            := Ada.Command_Line.Argument(Number => K);

            Position := Position + Command_Line_Length;

            --Ada.Text_Io.Put_Line(" cmd line "
            --& Integer'Image(K)
            --& " >"
            --& Ada.Command_Line.Argument(Number => K)
            --& '<'
            --& Integer'Image(Command_Line_Length)
            --& Chaine(1 .. Position)
            --);
         end loop;
         Position := - 1 + Position;

      end if;
      -- debug

      Ada.Text_Io.Put_Line("Data_IO.Data_Entry debug chaine =>"
                           & Chaine(1 .. Position)
                           & '<');

      return new String'(Chaine(1 .. Position));

   end Data_Entry;
```

15.4.4 Le module test de la validité

15.4.4.1 Définition des procédure et fonctions

```
Function valid_user_input(user_input: in string_access_type) return boolean;
```

15.4.4.2 Traitement

15.4.4.2.1 Introduction: la grammaire d'une expression arithmétique

Le test de validité se fera en utilisant une grammaire.

Rappel : la grammaire d'une expression arithmétique en notation infix, avec prise en compte de la dualité signe/opérateur pour + et -, peut s'écrire :

```
EXPRESSION      ::= [ ADD_OPERATOR ] TERM { ADD_OPERATOR TERM}
TERM              ::= NUMBER { MULT_OPERATOR [ADD_OPERATOR] NUMBER}
ADD_OPERATOR    ::= + | -
MULT_OPERATOR   ::= * | /
NUMBER            ::= DIGIT { DIGIT } [ . DIGIT { DIGIT } ]
DIGIT             ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Rappel : Les abréviations (généralement utilisées en algorithmique pour décrire une grammaire) sont les suivantes :

{ xxxxxx **}**	veut dire	0 à n répétitions de xxxxxx
[yyyyyyy **]**	veut dire	yyyyyy est optionnel
|	veut dire	**ou**

15.4.4.2.2 Traduction de la grammaire dans un automate à états finis

15.4.4.2.3 Les transitions

Une lecture attentive de la grammaire permet de faire la liste des *symboles terminaux*, qui ne peuvent pas se décomposer davantage: ce sont les transitions. Tous les caractères du clavier doivent être pris en compte.

1. digits — les caractères 0 1 2 3 4 5 6 7 8 9
2. add_operator — les caractères + -
3. mult_operator — les caractères * /
4. decimal_point — le caractère .
5. Miscellaneous — tous les autres caractères du clavier
6. Space — le caractère espace

15.4.4.2.4 La table de traduction caractères ==> transitions

Chaque caractère de l'entrée utilisateur sera scanné un par un et pour utiliser l'automate il faut affecter chaque caractère à une transition.
Le faire avec des tests n'est pas une bonne idée car les « if » imbriqués seraient très difficilement lisibles. Nous le ferons avec une table de vérité.

- les caractères 0 1 2 3 4 5 6 7 8 9 — digits
- les caractères + - — add_operator
- les caractères * / — mult_operator
- le caractère . — decimal_point
- le caractère espace — space
- tous les autres caractères du clavier — Miscellaneous

Caractères => Transitions	
0 1 2 3 4 5 6 7 8 9	digits
+ -	add_operator
* /	mult_operator
.	decimal_point
espace	space
Tous les autres caractères	Miscellaneous

Nous déclarerons donc un tableau :
la démarche est toujours la même : déclarer le **type du contenu**, déclarer le **type de l'indice**, déclarer le **type du tableau**,déclarer la **variable de type tableau** et **initialiser le contenu** de la variable.

1) déclarer le type pour le contenu

   contenu => transition

   le type est une liste de noms (add_operator, digits,...) en ada c'est une énumération
   voir cours chapitre 3.2.7:

```
type Transition_Type is(Digit,
                        Add_Operator,
                        Mult_Operator,
                        Decimal_Point,
                        Space,
                        Miscellaneous);
```

2) déclarer le type pour l'indice du tableau

indice => caractères

Le **caractère** est un type prédéfini , nous avons besoin de tous les caractères sans restriction => **aucun besoin de déclaration**. Le type **character** sera utilisé.

3) déclarer le type du tableau

Voir le cours chapitre 3.2.8

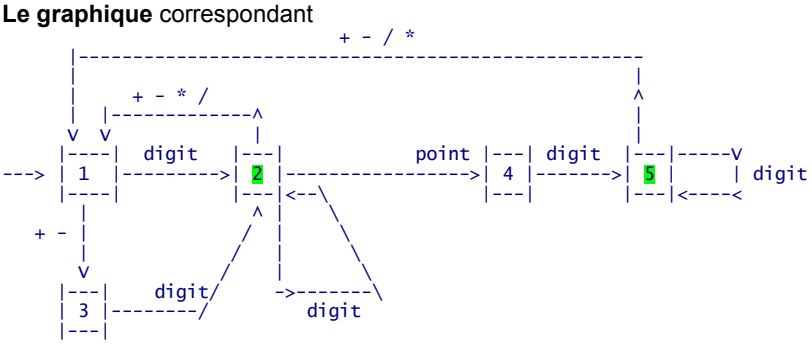
```
type Character_To_Transition_Table_Type  is array(Character'Range) of Transition_Type;
```

4) déclarer la variable de type tableau et initialiser le contenu en respectant ce qui est illustré ci-dessus

```
Character_To_Transition_Table : constant Character_To_Transition_Table_Type
       := ('0' .. '9'=> Digit,
           '+'|'-'   => Add_Operator,
           '*'|'/'   => Mult_Operator,
           '.'       => Decimal_Point,
           ' '       => Space,
          others     => Miscellaneous);
```

La ligne '0'..'9' signifie tous les caractères de 0 à 9
la ligne '+'|'-' signifie le caractère + et le caractère – la barre verticale | signifie et
Noter l'emploi de « others » qui permet d'associer tous les autres caractères à Miscellaneous

Le graphique correspondant



Les états valables sont 2 et 5

15.4.4.2.5 La matrice états transitions

C'est la traduction directe du graphique. **Chaque case de la matrice contient le numéro de l'état suivant** pour la combinaison état / transition.

	états							
transitions		**0**	**1**	**2**	**3**	**4**	**5**	
	Digit	0	2	2	2	5	5	
	Add_Operator	0	3	1	0	0	1	
	Mult_Operator	0	0	1	0	0	1	
	Decimal_Point	0	0	4	0	0	0	
	Miscellaneous	0	0	0	0	0	0	
	Space	0	1	2	3	4	5	

Les **zéros correspondent** à des combinaisons états transitions non définies par la grammaire et par conséquent **à des erreurs**.
Remarquer:

- L'usage de l'état zéro pour figer le processus dès qu'une erreur est rencontrée. Un fois dans cet état l'automate n'en sort pas!
- Que la transition « space » ne modifie pas l'état de l'automate, l'espace ne change rien et n'a pas d'influence.
- Les états valables en sortie de l'automate sont 2 et 5

- Déclaration du type des deux indices
  Le type transition est déjà défini plus haut
  Une variable de type transition peut se déclarer comme suit:

```
Transition        : Transition_Type;
```

- Le type état de l'automate peut se définir comme:
  `type State_Type is new Integer range 0 .. 5;`
  la variable correspondante peut se déclarer comme :
  `State             : State_Type := 1;`
  Elle est initialisée à 1 ce qui est indispensable l'automate commence à l'état 1

- Déclaration du type de contenu
  Le contenu de la matrice est un état déjà déclaré plus haut

- Déclaration du type de la matrice

```
type State_Transition_Matrix_Type
    is array(Transition_Type'Range, State_Type'Range)
    of
    State_Type;
```

- Déclaration de la variable (matrice) et son initialisation

Remarquer l'initialisation:

Elle est lisible

Le mot réservé « constant » qui protège contre toute modification ultérieure

La ligne commentée – S T A T E S ..... qui permet de mettre des titres aux colones

```
State_Transition_Matrix : constant State_Transition_Matrix_Type
      -- S T A T E S        0  1  2  3  4  5
      := (Digit         => (0, 2, 2, 2, 5, 5),
          Add_Operator  => (0, 3, 1, 0, 0, 1),
          Mult_Operator => (0, 0, 1, 0, 0, 1),
          Decimal_Point => (0, 0, 1, 0, 0, 0),
          Space         => (0, 1, 2, 3, 4, 5),
          Miscellaneous => (0, 0, 0, 0, 0, 0));
```

- Declaration du caractère courant:

```
   Current_Character : Character;
```

15.4.4.2.6 Le fonctionnement de l'automate

pour traiter chaque caractère on utilisera une boucle sur le nombre de caractères à traiter.

Initialiser l'état à 1:	Fait à la déclaration	
boucle pour chaque caractère:	for K in User_Input.all'Range loop	
Lecture du caractère	Current_Character := User_Input.all(K);	
traduire en transitions avec la table de vérité	Transition:= Character_To_Transition_Table	(Current_Character);
faire état := matrice(transition,état);	State:=State_Transition_Matrix(Transition,	State);
fin boucle	end loop;	
la grammaire est correcte si état = 2 ou 5.	return state=2 or else state=5;	

Le code est très simple et c'est toujours le même pour tous les automates à états finis. Seuls changent la table de traduction et la matrice.

15.4.4.3 Le code complet

```
function Valid_User_Input(User_Input : in     Data.String_Access_Type)
      return Boolean
      is

      type Transition_Type is(Digit,
                              Add_Operator,
                              Mult_Operator,
                              Decimal_Point,
                              Space,
                              Miscellaneous);

      type Character_To_Transition_Table_Type
      is array(Character'Range) of Transition_Type;

      Character_To_Transition_Table : constant Character_To_Transition_Table_Type
      := ('0' .. '9'=> Digit,
          '+'|'-'   => Add_Operator,
          '*'|'/'   => Mult_Operator,
          '.'       => Decimal_Point,
          ' '       => Space,
          others    => Miscellaneous);


      type State_Type is new Integer range 0 .. 5;

      type State_Transition_Matrix_Type
      is array(Transition_Type'Range, State_Type'Range)
      of
```

```
      State_Type;

      State_Transition_Matrix : constant State_Transition_Matrix_Type
      -- S T A T E S        0  1  2  3  4  5
      := (Digit         => (0, 2, 2, 2, 5, 5),
          Add_Operator  => (0, 3, 1, 0, 0, 1),
          Mult_Operator => (0, 0, 1, 0, 0, 1),
          Decimal_Point => (0, 0, 1, 0, 0, 0),
          Space         => (0, 1, 2, 3, 4, 5),
          Miscellaneous => (0, 0, 0, 0, 0, 0));

      State             : State_Type := 1;
      Transition        : Transition_Type;
      Current_Character : Character;

      begin

      for K in User_Input.all'Range loop

         Current_Character := User_Input.all(K);

         Transition := Character_To_Transition_Table(Current_Character);

         State := State_Transition_Matrix(Transition, State);

         Ada.Text_Io.Put_Line("Validation.Valid_User_Input k"
                              & Integer'Image(K)
                              & " char "
                              & Current_Character
                              & " transition "
                              & Transition_Type'Image(Transition)
                              & " state "
                              & State_Type'Image(State));
      end loop;

      Ada.Text_Io.Put_Line("Validation.Valid_User_Input "
                           & Boolean'Image(State = 2 or else State = 5));

      return State = 2 or else State = 5;
   end Valid_User_Input;
```

15.4.5 Le Module découpage en opérateur opérande

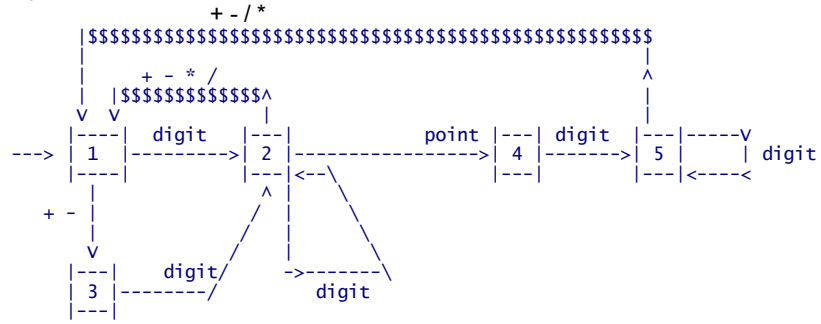
15.4.5.1 Définition des procédure et fonctions

Function parse (user_input : in string_access_type) return string_access_table_type;

15.4.5.2 Traitement

15.4.5.2.1 Le résumé

Il y a plusieurs solutions, l'**instrumentation de l'automate précédent** sera mise en oeuvre:



Remarquez que les transitions marquées avec les signes $ correspondent :

A la **fin d'un opérande**
Ont lieu quand un **opérateur** est le **caractère courant**
cela permettra un découpage (parsing) facile :

Réutilisation du code précédent avec ajout d'un test et du stockage successif des opérateurs et opérandes.

Ce test n'étant pas simple sera réalisé avec un table de vérité.

	Etats							
transitions		**0**	**1**	**2**	**3**	**4**	**5**	
	Digit	faux	faux	faux	faux	faux	faux	
	Add_Operator	faux	faux	vrai	faux	faux	vrai	
	Mult_Operator	faux	faux	vrai	faux	faux	vrai	
	Decimal_Point	faux	faux	faux	faux	faux	faux	
	Miscellaneous	faux	faux	faux	faux	faux	faux	
	Space	faux	faux	faux	faux	faux	faux	

La boucle de fonctionnement de l'automate est la même que pour le test grammatical. Un test est introduit avant la ligne qui fait avancer l'automate. Ceci pour pouvoir tester l'état courant et la transition courante. En utilisant la table de vérité ci-dessus.

Si le test est vrai
c'est la fin de l'opérande courant => il faut le stocker dans le tableau de sortie.
Qui dit stock dans un tableau dit **indice courant** à conserver et incrémenter : il faut donc une variable.
Le caractère courant est l'opérateur => il faut le stocker dans le tableau de sortie.
Incrémenter l'indice du tableau.

l'opérateur a toujours une longueur de 1 et c'est toujours le caractère courant:
il n'y a aucun besoin de conserver le début.

L'opérande a une longueur variable depuis la position suivant celle du dernier opérateur jusqu'à la **position du caractère courant** -1. Il faut donc une variable pour stocker la valeur de la **dernière position stockée**.

Le cas du premier opérande et du dernier opérande sont à traiter séparément.
Le premier:
Il se traite en initialisant la variable de position à 1;
Le dernier:
Après la fin de la boucle de l'automate, il faut pouvoir disposer de la position du caractère courant. Dans le code du test grammatical que nous réutiliseront ici en grande partie c'était l'indice de la boucle « for » . Il faut changer cette boucle « for » en boucle « while », créer une variable pour stoker l'indice courant, et l'incrémenter à l'intérieur de la boucle comme dernière instruction avant le « end if; ». De cette façon cette variable sera toujours disponible après la fin de la boucle.

15.4.5.2.2 Les détails

15.4.5.2.3 Parties copiées et collées du module précédent

Les parties communes avec le module précédent ne seront pas commentées.

15.4.5.2.4 Les variables

La fonction renvoie une variable de type Data.String_Access_Table_Type. Il faut créer une variable de ce type, l'initialiser à vide si nécessaire ici un pointeur vide représenté pa « null », la remplir pendant le fonctionnement et la renvoyer par return xxx; à la fin. Voici la déclaration:

```
Dummy : Data.String_Access_Table_Type := (others => null);
```

k comme indice de boucle

```
K                : Integer := 1;
```

how_many_tokens comme indice courant du tableau de sortie {ici dummy}.

```
How_Many_Tokens   : Natural := 0;
```

end_of_token comme début de l'opérande, initialisé au premier indice du tableau {'first}.

```
End_Of_Token      : Natural := User_Input.all'First;
```

15.4.5.2.5 La table de vérité

Cette table permet: p**endant le fonctionnement de l'automate** et **avant de faire passer l'automate dans l'état suivant**, de **tester si le caractère courant est un opérande**.

Il faut, comme d'habitude, type des indices, type du contenus et type de la matrice.
Ici tout est déjà déclaré: état, transition sont les mêmes que pour l'automate et le contenu est un booléen qui est un type prédéfini du langage.

```
type End_Of_Token_Matrix_Type
is array(Transition_Type'Range, State_Type'Range)
of Boolean;

End_Of_Token_Matrix : constant End_Of_Token_Matrix_Type
-- S T A T E S       0      1      2     3      4      5
:= (Digit         => (False, False, False, False, False, False),
    Add_Operator  => (False, False, True, False, False, True),
    Mult_Operator => (False, False, True, False, False, True),
    Decimal_Point => (False, False, False, False, False, False),
    Space         => (False, False, False, False, False, False),
    Miscellaneous => (False, False, False, False, False, False));
```

15.4.5.2.6 Les détails du déroulement

Une boucle while s'écrira : tant que k est compris entre le premier et le dernier indice de l'entrée utilisateur exécuter la boucle. Ada nous permet plusieurs raccourcis différents en utilisant les attributs first last ou range

```
while K in User_Input'first .. User_Input'first  loop
```

Ou, ce qui est équivalent :

```
while K in User_Input'Range loop
.....
......
k :=k+1;
end loop;
```

Complétons la boucle avec ce qui proviens du code du module précédent et instrumentons la boucle:

```
while k in ...........   loop

Current_Character := User_Input.all(K);

Transition := Character_To_Transition_Table(Current_Character);
```

Ici, AVANT le passage de l'automate à l'état suivant on teste si c'est un opérande

```
State := State_Transition_Matrix(Transition, State);

K := 1 + K;

end loop;
```

Le test s'écrit simplement :

```
if End_Of_Token_Matrix(Transition, State)
then
.....
.....
end if;
```

Voyons les détails du fonctionnement avec
un exemple simple
k comme indice de boucle
dummy comme nom du tableau de sortie
how_many_tokens comme indice courant du tableau de sortie
end_of_token comme début de l'opérande.

```
test positif
  |  |
  v  v
22+33*-44
```

L'automate se déroulera comme précédemment et le test opérateur tel que décrit ci-dessus sera positif comme indiqué avec k égal 3 puis k=7.
Quant un opérateur est détecté {le test renvoie vrai}, il faut
Incrémenter l'indice du tableau
stocker l'opérande précédent qui commence au caractère 1 et fini au caractère k-1 dans la structure de sortie (ici un tableau avec indice)

```
How_Many_Tokens := 1 + How_Many_Tokens;

Dummy(How_Many_Tokens):= new String'(User_Input.all(End_Of_Token .. K - 1));
```

stoker l'opérateur courant qui a une longueur de **UN** (k..k) dans la structure de sortie

```
How_Many_Tokens := 1 + How_Many_Tokens;

Dummy(How_Many_Tokens) := new String'(User_Input.all(K .. K));
```

mettre dans une variable la position ou commence l'opérande suivant (qui existe obligatoirement puisque la grammaire est correcte)

```
End_Of_Token := 1 + K;
```

A la fin de la boucle, le dernier opérande n'est pas stocké , il faut le faire et c'est là que nous utilisons après la fin de boucle l'indice k de la boucle. C'est pour cette ligne que nous utilisons une boucle « while » et non pas une boucle « for »

```
How_Many_Tokens := 1 + How_Many_Tokens;

Dummy(How_Many_Tokens) := new String'(User_Input.all(End_Of_Token .. K - 1));
```

Maintenant il ne reste plus qu'a renvoyer le tableau :

```
Return dummy;
```

15.4.5.3 Le code complet

```
function Parse(User_Input : in     Data.String_Access_Type)
   return Data.String_Access_Table_Type
   is

   Dummy : Data.String_Access_Table_Type;

   type Transition_Type is(Digit,
                           Add_Operator,
                           Mult_Operator,
                           Decimal_Point,
                           Space,
                           Miscellaneous);

   type Character_To_Transition_Table_Type
   is array(Character'Range) of Transition_Type;

   Character_To_Transition_Table : constant Character_To_Transition_Table_Type
   := ('0' .. '9'=> Digit,
       '+'|'-'   => Add_Operator,
       '*'|'/'   => Mult_Operator,
       '.'       => Decimal_Point,
       ' '       => Space,
       others    => Miscellaneous);


   type State_Type is new Integer range 0 .. 5;

   type State_Transition_Matrix_Type
   is array(Transition_Type'Range, State_Type'Range)
   of
   State_Type;
```

```
   State_Transition_Matrix : constant State_Transition_Matrix_Type
   -- S T A T E S        0  1  2  3  4  5
   := (Digit         => (0, 2, 2, 2, 5, 5),
       Add_Operator  => (0, 3, 1, 0, 0, 1),
       Mult_Operator => (0, 0, 1, 0, 0, 1),
       Decimal_Point => (0, 0, 1, 0, 0, 0),
       Space         => (0, 1, 2, 3, 4, 5),
       Miscellaneous => (0, 0, 0, 0, 0, 0));


   type End_Of_Token_Matrix_Type
   is array(Transition_Type'Range, State_Type'Range)
   of Boolean;

   End_Of_Token_Matrix : constant End_Of_Token_Matrix_Type
   -- S T A T E S        0      1      2     3      4      5
   := (Digit         => (False, False, False, False, False, False),
       Add_Operator  => (False, False, True, False, False, True),
       Mult_Operator => (False, False, True, False, False, True),
       Decimal_Point => (False, False, False, False, False, False),
       Space         => (False, False, False, False, False, False),
       Miscellaneous => (False, False, False, False, False, False));

   State             : State_Type := 1;
   Transition        : Transition_Type;
   Current_Character : Character;
   End_Of_Token      : Natural := User_Input.all'First;
   How_Many_Tokens   : Natural := 0;
   K                 : Integer := 1;

   begin

   while K in User_Input'Range loop

      Current_Character := User_Input.all(K);

      Transition := Character_To_Transition_Table(Current_Character);

      if End_Of_Token_Matrix(Transition, State)
         then

         How_Many_Tokens := 1 + How_Many_Tokens;

        Dummy(How_Many_Tokens):= new String'(User_Input.all(End_Of_Token .. K - 1));

         How_Many_Tokens := 1 + How_Many_Tokens;

         Dummy(How_Many_Tokens) := new String'(User_Input.all(K .. K));

         End_Of_Token := 1 + K;

      end if;

      State := State_Transition_Matrix(Transition, State);


      K := 1 + K;

   end loop;


   How_Many_Tokens := 1 + How_Many_Tokens;

   Dummy(How_Many_Tokens) := new String'(User_Input.all(End_Of_Token .. K - 1));

   return Dummy;
end Parse;
```

15.4.6 Le module traduction du découpage précédent en postfix

C'est la partie la plus difficile, elle vous permet de traduire un algorithme, trouvé dans la littérature en code. l'algorithme de traduction fait appel à un stack. C'est une structure munie d'opérations que l'auteur du livre admet être connu de l'utilisateur. Le stack sera explicité dans son fonctionnement théorique et un code simple sera écrit.

15.4.6.1 Définition des procédure et fonctions

```
function infix_to_postfix ( from : in  Data.string_access_table_type)
return data.string_access_table_type;
```

15.4.6.2 Traitement

Exemples :

2+3-4-5	= 2 3 4 5--+	= 6
2-3+4+5	= 2 3 4 5++-	= -10
2+3-4+5	= 2 3 4 5+-+	= -4
2-3+4-5	= 2 3 4 5-+-	= 0
2*3+4/5	= 2 3* 4 5/+	= 6.8
2+3/4-5	= 2 3 4/ 5-+	= -2.25
2*3/4*5	= 2 3* 4/ 5*	= 7.5
2/3/4/5	= 2 3/ 4/ 5/	= 0.0333333333
2/3*4/5	= 2 3/ 4* 5/	= 0.5333333333
2+3-4*5	= 2 3 4 5*-+	= -15

15.4.6.3 Nature des données d'entrée de l'algorithme

1. L'algorithme ci-dessous n'est valable que si l'expression infix RESPECTE la grammaire.

2. Le découpage de l'entrée utilisateur en opérateur et opérande en éléments est supposé déjà fait (c'est un autre algorithme).

3. La priorité des opérateurs est bien entendu : * et / > + et - > signe spécial { voir algorithme ci-dessous}.

4. L'associativité des opérateurs est prise en compte.

15.4.6.4 Algorithme de Traduction

Pour cette section vous allez traduire un algorithme pré-existant en code (voir plus bas).
L'algorithme fait référence à un « stack » qui est une structure munie d'opérations: un stack fonctionne selon le principe **dernier entré => premier sorti** (last in first out).

15.4.6.4.1 Le stack

Un stack possède un « **sommet** » qui est la valeur la plus récente introduite dans le « stack », c'est la s**eule valeur accessible**! Il fonctionne selon le principe **dernier entré premier sorti**. C'est cette propriété qui justifie son utilisation car elle **INVERSE L'ODRE D'ENTREE**.

Cette structure va être implémentée spécialement avec les opérations suivantes:

1. **push** introduit une valeur au sommet du stack.
2. **pop** enlève la valeur du sommet du stack.
3. **Value** renvoie la valeur du sommet du stack.
4. **clear** initialise le stack et « vide le contenu si nécessaire ».

Nous utiliseront un stack très simplifié qui permettra de coder des routines internes à une procédure / fonction. Leur position dans le code est indiquée ci-dessous

```
function infix_to_postfix ( from : in  string_access_table_type)
is
les routines du gestion du stack seront introduites ici
begin
.....
```

Les données du stack seront stockées dans un tableau (de même nature que les données d'entrée: data.string_access_table_type) avec un indice qui conservera la position du sommet.
Il faut créer les variables correspondantes :
top_of_stack un entier de 0 à ...
stack un tableau de type data.string_access_table_type initialisé vide

- Procedure push (data :in data.string_access_type)

  incrémenter top_of_stack.
  mettre data dans le tableau à l'indice top_of_stack.

- function value return data.string_access_type

  renvoie le contenu du tableau à l'indice top_of_stack.

- procedure pop

  décrémente l'indice top_of_stack.

- procedure clear

  met l'indice top_of_stack à zéro.

15.4.6.4.2 Priorité Des Opérateurs

La priorité des opérateurs va influencer le déroulement du programme. La table de vérité de la comparaison :

opérateur courant **<** opérateur du sommet du stack
s'écrit comme suit *{ le signe spécial, introduit dans le stack au début, est une astuce qui sert à ne **JAMAIS** comparer un stack vide avec un opérateur, les parenthèses sont traitées comme le signe spécial }*. La table de vérité prends en compte l'associativité des opérateurs :

24/3/4 signifie (24/3)/4
24/3*4 signifie (24/3)*4

Table de vérité de la comparaison opérateur courant < opérateur du sommet du stack.
Si vrai, il y a sortie de la boucle qui sert à vider le contenu du stack d'opérateurs vers la sortie

	Opérateur du sommet du stack					
Opérateur Courant		*	/	+	-	**Signe Spécial**
	*	True	False	True	True	True
	/	False	False	True	True	True
	+	False	False	True	True	True
	-	False	False	True	True	True
	Signe Spécial	False	False	False	False	True

L'implémentation de la priorité des opérateurs se fait en deux étapes: la traduction opérateur en symboles (pour conserver un taille raisonnable à la table de vérité) et la lecture de la table de vérité.

15.4.6.5 Partie principale

L'algorithme très simplifié dans la littérature (traduit de l'anglais) est comme suit :

Initialiser le stack qui contiendra les opérateurs.

Mettre le **marqueur spécial** de priorité la plus faible possible dans le stack
Boucle principale

Lecture de l'élément suivant (opérande, opérateur ou fin de l'expression à traduire)

{L'action dépends du type de l'élément}

1. **L'élément est un opérande**

Envoyer cet élément en sortie

2. **L'élément est un opérateur**

1. Boucle
2. Sortie de la boucle si l'élément a une priorité < que le sommet du stack : *{élément < opérateur du sommet du stack ou le signe spécial si le stack ne contient pas d'opérateur}*
3. Envoyer le sommet du stack en sortie
4. Pop stack
5. Fin boucle
6. Mettre l'élément *{ c'est le dernier opérateur lu }* dans le stack

3. **Il n'y a plus d'élément à lire**

1. Sortie de la boucle principale

Fin boucle principale

4. **Vider les opérateurs restant dans le stack**

1. Boucle
2. Sortie de la boucle si le sommet du stack est le signe spécial
3. Envoyer le sommet du stack en sortie
4. Pop stack
5. Fin boucle

15.4.6.6 L'algorithme plus détaillé

Clear stack

Push signe spécial dans le stack

```
boucle sur tous les tokens possibles

        lire le token courant dans la variable token
        sortie boucle si le token courant  est vide
                si token est un opérande
                alors
                        remplir la position courante en sortie avec ce token


                sinon (le token est  un opérateur)
                        boucle
                                mettre le sommet du stack dans la variable sommet_du_stack
                                sortie de boucle si  Token < sommet du stack
                                remplir la position courante en sortie avec sommet_du_stack
                                Pop stack
                        fin boucle
                        push token
                fin si
        fin boucle
fin boucle


boucle tant que sommet du stack est différent du signe spécial
        remplir la position courante en sortie avec sommet_du_stack
        Pop
fin boucle
```

15.4.6.7 Analyse de l'algorithme détaillé

L'entrée est un tableau de type Data.String_Access_Table_Type
la sortie est un tableau de type Data.String_Access_Table_Type
Il faut créer un index de tableau: une variable pour la lecture de l'entrée et une variable pour l'écriture en sortie, nous utiliserons le type de variable déjà définies pour l'index des tableaux entrée/sortie :
Il faut aussi un signe spécial compatible avec le type des données contenues dans le stack d'opérateurs.
Il faut une variable pour le token courant et le sommet du stack d'opérateurs.

15.4.6.8 Le code du stack

C'est le premier exemple de routine interne à un autre routine. Il est donc situé entre is et begin.

15.4.6.8.1 Les variables

Une version extrêmement simple est utilisée:
un tableau pour stocker les valeurs.

```
Top_Of_Stack : Integer range 0 .. Data.Table_Size'Last := 0;
```

un indice pour indiquer le sommet.

```
Stack        : Data.String_Access_Table_Type          := (others => null);
```

15.4.6.8.2 Le code

15.4.6.8.3 Push

Introduit une valeur au sommet du stack: stock de la valeur et incrémentation de l'indice.

```
procedure Push(The_Item : in     Data.String_Access_Type)
 is
 begin
 Top_Of_Stack        := Top_Of_Stack + 1;
 Stack(Top_Of_Stack) := The_Item;
end Push;
```

15.4.6.8.4 Pop

Enlève la valeur du sommet du stack. (Rend invisible la valeur précédente sans l'effacer).

```
procedure Pop
   is
   begin
   Top_Of_Stack := Top_Of_Stack - 1;
end Pop;
```

15.4.6.8.5 Value

Renvoie la valeur du sommet du stack.

```
function Value return Data.String_Access_Type
   is
   begin
   return Stack(Top_Of_Stack);
end Value;
```

15.4.6.8.6 Clear

Initialise le stack et « vide le contenu si nécessaire ». Ici il rend invisible tout le contenu mais n'efface rien.

```
procedure Clear
   is
```

```
   begin
   Top_Of_Stack := 0;
end Clear;
```

15.4.6.9 le code du test de priorité des opérateurs

Deux notions nouvelles vont etre introduite à cette occasion :

- Overloading
  C'est le fait d'avoir plus d'une fonction ou procédure avec les mêmes noms et les memes paramètres. La différence se situe au niveau du type des paramètres.

- Création d'une fonction dont le nom est un opérateur prédéfini.

Pour que le code soit lisible aussi naturellement que possible, je veux utiliser le signe « < » pour signifier plus petit pour comparer les priorités des opérateurs.

15.4.6.9.1 Les variables

L'entrée se fait par Data.String_Access_Type pour les deux cotés de la comparaison.
Il faut quelque part les transformer en opérateurs puis utiliser la table de vérité. Nous utiliseront une fonction interne à la fonction "<" .
Cette fonction « Operator_Type_Of » traduira en un opérateur de type « Operator_Type » qui permettra un code LISIBLE

```
function "<" (Left  : in Data.String_Access_Type;
              Right : in     Data.String_Access_Type)
          return Boolean
is

          type Operator_Type is(Multiply,
                                Divide,
                                Plus,
                                Minus,
                                End_Of_Expression_Token);

          function Operator_Type_Of(The_Token : in     Data.String_Access_Type)
             return Operator_Type
             is
.......

end  Operator_Type_Of;
........
end "<";
```

Une table de vérité (matrice avec ligne et colones de type « Operator_Type » contenant des booléens.

```
type Truth_Table_Type is array(Operator_Type, Operator_Type) of Boolean;

          Truth_Table : constant Truth_Table_Type
          --                                      sommet du stack
          --    token                        *     /     +     -    EOT
          := (Multiply               => (True, False, True, True, True),
              Divide                 => (False, False, True, True, True),
              Plus                   => (False, False, True, True, True),
              Minus                  => (False, False, True, True, True),
              End_Of_Expression_Token => (False, False, False, False, True));
```

Le reste du code est très lisible car décomposé complètement.

15.4.6.9.2 Le code

```
       function "<" (Left  : in Data.String_Access_Type;
                     Right : in     Data.String_Access_Type)
          return Boolean
          is
          Result : Boolean;

          type Operator_Type is(Multiply,
                                Divide,
                                Plus,
                                Minus,
                                End_Of_Expression_Token);


          function Operator_Type_Of(The_Token : in     Data.String_Access_Type)
             return Operator_Type
             is

             Operator : Operator_Type := Operator_Type'Last;
             begin

             if The_Token.all = "+"
                then
                Operator := Plus;
                elsif The_Token.all = "-"
```

```
            then
            Operator := Minus;
            elsif The_Token.all = "/"
            then
            Operator := Divide;
            elsif The_Token.all = "*"
            then
            Operator := Multiply;
            else
            Operator := End_Of_Expression_Token;
         end if;

         return Operator;

      end Operator_Type_Of;

      type Truth_Table_Type is array(Operator_Type, Operator_Type) of Boolean;

      Truth_Table : constant Truth_Table_Type
      --                                        sommet du stack
      --    token                          *     /     +     -    EOT
      := (Multiply               => (True, False, True, True, True),
          Divide                 => (False, False, True, True, True),
          Plus                   => (False, False, True, True, True),
          Minus                  => (False, False, True, True, True),
          End_Of_Expression_Token => (False, False, False, False, True));

      Left_Operator  : Operator_Type;
      Right_Operator : Operator_Type;

      begin

      Left_Operator := Operator_Type_Of(The_Token => Left);

      Right_Operator := Operator_Type_Of(The_Token => Right);

      Result := Truth_Table(Left_Operator, Right_Operator);

      return Result;
end "<";
```

15.4.6.10 Le code du test opérande

Le test grammatical étant supposé fait, un opérande n'est simplement pas un opérateur:

```
function Is_Operand(X : in     Data.String_Access_Type)
return Boolean
is
begin

 return   X.all /= "+"
 and then X.all /= "-"
 and then X.all /= "*"
 and then X.all /= "/";
end Is_Operand;
```

15.4.6.11 Le code du programme principal

Le code complet :

```
   function Infix_To_Postfix(From : in     Data.String_Access_Table_Type)
      return Data.String_Access_Table_Type
      is
      --$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
      --          V E R Y   S I M P L E    S T A C K
      --$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

      Top_Of_Stack : Integer range 0 .. Data.Table_Size'Last := 0;
      Stack        : Data.String_Access_Table_Type           := (others => null);

      procedure Push(The_Item : in     Data.String_Access_Type)
         is
         begin
         Top_Of_Stack        := Top_Of_Stack + 1;
         Stack(Top_Of_Stack) := The_Item;
      end Push;

      function Value return Data.String_Access_Type
         is
         begin
         return Stack(Top_Of_Stack);
      end Value;

      procedure Pop
         is
         begin
         Top_Of_Stack := Top_Of_Stack - 1;
      end Pop;

      procedure Clear
```

```
   is
   begin
   Top_Of_Stack := 0;
end Clear;


--$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
--                    O P E R A T O R   P R I O R I T Y
--$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$


--|              |_____________________|
--|              |  "<"                |
--|              |_____________________|
--|
function "<" (Left  : in Data.String_Access_Type;
              Right : in     Data.String_Access_Type)
   return Boolean
   is
   Result : Boolean;

   type Operator_Type is(Multiply,
                         Divide,
                         Plus,
                         Minus,
                         End_Of_Expression_Token);

   --|
   --|              |____________________|
   --|              | Operator_Type_Of   |
   --|              |____________________|
   --|
   function Operator_Type_Of(The_Token : in     Data.String_Access_Type)
      return Operator_Type
      is

      Operator : Operator_Type := Operator_Type'Last;
      begin
      --|
      --|  convert  to operator_type
      --|
      if The_Token.all = "+"
         then
         Operator := Plus;
         elsif The_Token.all = "-"
         then
         Operator := Minus;
         elsif The_Token.all = "/"
         then
         Operator := Divide;
         elsif The_Token.all = "*"
         then
         Operator := Multiply;
         else
         Operator := End_Of_Expression_Token;
      end if;

      return Operator;

   end Operator_Type_Of;
   --|
   --|
   --|
   --|
   type Truth_Table_Type is array(Operator_Type, Operator_Type) of Boolean;

   Truth_Table : constant Truth_Table_Type
   --                                        sommet du stack
   --    token                          *     /     +     -    EOT
   := (Multiply               => (True, False, True, True, True),
       Divide                 => (False, False, True, True, True),
       Plus                   => (False, False, True, True, True),
       Minus                  => (False, False, True, True, True),
       End_Of_Expression_Token => (False, False, False, False, True));

   Left_Operator  : Operator_Type;
   Right_Operator : Operator_Type;

   begin
   --|
   --|  convert left to operator_type
   --|
   Left_Operator := Operator_Type_Of(The_Token => Left);
   --|
   --|  convert right to operator_type
   --|
   Right_Operator := Operator_Type_Of(The_Token => Right);
   --|
   --|operator precedence reading from truth table
   --|
   Result := Truth_Table(Left_Operator, Right_Operator);
   --|
   --|return operator priority
   --|
   return Result;
end "<";
```

```
--$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
--                    O P E R A N D E    ?????
--$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

--|              |____________________|
--|              |  Is_Operand        |
--|              |____________________|
--|
function Is_Operand(X : in     Data.String_Access_Type)
   return Boolean
   is
   begin
   --|
   --|  simple test only four valid possibilities
   --|
   return X.all /= "+"
   and then X.all /= "-"
   and then X.all /= "*"
   and then X.all /= "/";
end Is_Operand;

--$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
--                    variables for the main
--$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

Special_Sign   : constant Data.String_Access_Type := new String'("$");
Program_Output : Data.String_Access_Table_Type    := (others => null);

Token                 : Data.String_Access_Type                := null;
Top_Of_Operator_Stack : Data.String_Access_Type                := null;
Index                 : Integer range 0 .. Data.Table_Size'Last := 0;
use type Data.String_Access_Type;
begin
--|
--|  init
--|
Clear;
--|
--|Push(operator_stack,end_of_expression_token)
--|
Push(The_Item => Special_Sign);
--|
--|boucle sur tous les tokens possibles
--|
for K in From'Range loop
   --|
   --|read current token
   --|
   Token := From(K);
   --|
   --|done => exit
   --|
   exit when Token = null;
   --|
   --| token = operand ??
   --|
   if Is_Operand(Token)
      then
      --|
      --|output (token)
      --|
      Index                 := 1 + Index;
      Program_Output(Index) := Token;
      --|
      --| else token = operator
      --|
      else
      --|
      --|while token < top(operator_stack) loop
      --|
      loop
         --|
         --|read top of of stack into Top_Of_Operator_Stack
         --|
         Top_Of_Operator_Stack := Value;
         --|
         --|exit if priority
         --|
         exit when Token < Top_Of_Operator_Stack;
         --|
         --|output  top operator_stack
         --|
         Index                 := 1 + Index;
         Program_Output(Index) := Top_Of_Operator_Stack;
         --|
         --|remove top token from stack
         --|
         Pop;
         --|
         --|end loop on operator priority
         --|
      end loop;
      --|
      --| push current token in stack
      --|
```

```
            Push(The_Item => Token);
            --|
            --|end if operator operande
            --|
         end if;
         --|
         --|end loop on tokens
         --|
      end loop;
      --|
      --| emptying the rest of the stack but not the end_of_line marker
      --|
      loop
         exit when Value = Special_Sign;
         --|
         --|Output top(operator_stack)
         --|
         Index                 := 1 + Index;
         Program_Output(Index) := Value;
         --|
         --|pop(operator_stack)
         --|
         Pop;
      end loop;
      --|
      --|clear operator_stack
      --|
      Clear;
      --|
      --|debug output
      --|
      for Z in 1 .. Index loop

         Ada.Text_Io.Put_Line("Traduction.Infix_To_Postfix Program_Output("
                              & Integer'Image(Z)
                              & ").all >"
                              & Program_Output(Z).all
                              & '<');

      end loop;
      --|
      --|all done
      --|
      return Program_Output;

   end Infix_To_Postfix;
```

15.4.7 Calculs à partir du postfix

15.4.7.1 Définition des procédure et fonctions

Function calcul_from_postfix (postfix : in string_access_table_type)
return float;

15.4.7.2 Traitement

15.4.7.2.1 Résumé

1. On dispose d 'un stack pouvant contenir les opérandes, vide au début.
2. On lit successivement tous les opérateurs et opérandes contenus dans la traduction posifix.
3. Si l'élément courant est un opérande on le met dans le stack
4. Si l'élément courant est un opérateur on extrait les deux derniers opérandes du stack, on effectue le calcul demandé par l'opérande et on remet le résultat dans le stack.
5. Quand il n'y a plus d'élément, le stack ne contient qu'une valeur, le résultat.

15.4.7.2.2 Exemple

Par exemple, pour calculer 2+3/4 en notation postfix : 2 3 4 / +

les étapes sont:

stack vide au départ

Sommet du stack	Vide
Partie cachée	Vide
Partie cachée	Vide

lecture de 2 c'est un opérande le mettre dans le stack

Sommet du stack	2
Partie cachée	Vide
Partie cachée	Vide

Lecture de 3 c'est un opérande le mettre dans le stack

Sommet du stack	3
Partie cachée	2
Partie cachée	Vide

Lecture de 4 c'est un opérande le mettre dans le stack

Sommet du stack	4
Partie cachée	3
Partie cachée	2

Lecture de / c'est un opérateur

x <- sommet du stack
pop stack

Sommet du stack	3
Partie cachée	2
Partie cachée	Vide

Et x =4

y <- sommet du stack
pop stack

Sommet du stack	2
Partie cachée	Vide
Partie cachée	Vide

Et x =4, y=3
calcul de z = y/x = ¾=0.75
Push z dans le stack

Sommet du stack	0.75
Partie cachée	2
Partie cachée	Vide

Lecture de + c'est un opérateur

x <- sommet du stack
pop stack

Sommet du stack	2
Partie cachée	vide
Partie cachée	Vide

Et x =0.75

y <- sommet du stack
pop stack

Sommet du stack	Vide
Partie cachée	Vide
Partie cachée	Vide

Et x =0.75, y=2
calcul de z = y+x = 2+ 0.75 = 2.75
Push z dans le stack

Sommet du stack	2.75
Partie cachée	Vide
Partie cachée	Vide

Fin de lecture la valeur est dans le sommet du stack.

15.4.7.2.3 Détails des opérations

vider le stack

```
Clear;
```

boucle sur le contenu postfix découpé en éléments successifs

```
while Postfix(Current_Index) /= null loop
```

élément courant = opérande :
oui

```
if Is_Operand(Postfix(Current_Index))
then
```

traduire opérande de chaîne de caractère en un réel **en ignorant les espaces**.

```
Number := Float'Value(Postfix(Current_Index).all);
```

push opérande dans le stack

```
Push(Number);
```

non

operand_1:= sommet du stack
pop stack

```
Operand_1 := Value;
             Pop;
```

operand_2:= sommet du stack
pop stack

```
Operand_2 := Value;
Pop;
```

test sur opérande

si + faire number:= operande_2 + operand_1

```
if Postfix(Current_Index).all = "+"
then
Number := Operand_2 + Operand_1;
```

si - faire number:= operande_2 - operand_1

```
elsif Postfix(Current_Index).all = "-"
then
Number := Operand_2 - Operand_1;
```

si * faire number:= operande_2 * operand_1

```
elsif Postfix(Current_Index).all = "*"
then
Number := Operand_2 * Operand_1;
```

si / faire number:= operande_2 / operand_1

```
elsif Postfix(Current_Index).all = "/"
then
Number := Operand_2 / Operand_1;
```

Plus d'autre choix :

```
end if;
```

mettre number dans le stack

```
Push(Number);
```

fin boucle

```
Current_Index := 1 + Current_Index;
end loop;
```

résultat:= sommet du stack;

```
return Value;
```

15.4.7.2.4 Le stack de réels

C'est le même que le stack d'opérateurs, utilisé dans la partie précédente, mais modifié pour stocker des réels. Les trois lignes ci-dessous sont modifiées (indiqué en italique et gras).

Stack : array(1 .. 50) of ***Float*** := (others => ***0.0***);
procedure Push(The_Item : in ***Float***) ....
function Value return ***Float*** .....

15.4.7.3 Le code complet

```
function Calcul_From_Postfix(Postfix : in     Data.String_Access_Table_Type)
       return Float
       is


       Top_Of_Stack : Integer range 0 .. Data.Table_Size'Last := 0;

       Stack : array(1 .. 50) of Float := (others => 0.0);

       procedure Push(The_Item : in     Float)
          is
          begin
          Top_Of_Stack        := Top_Of_Stack + 1;
          Stack(Top_Of_Stack) := The_Item;
       end Push;

       function Value return Float
          is
          begin
          return Stack(Top_Of_Stack);
       end Value;

       procedure Pop
          is
          begin
          Top_Of_Stack := Top_Of_Stack - 1;
       end Pop;

       procedure Clear
          is
          begin
          Top_Of_Stack := 0;
       end Clear;

       function Is_Operand(X : in     Data.String_Access_Type)
          return Boolean
          is
          begin
          return X.all /= "+"
          and then X.all /= "-"
          and then X.all /= "*"
          and then X.all /= "/";
       end Is_Operand;

       Current_Index : Data.Table_Size := Data.Table_Size'First;
       Number        : Float           := 0.0;
```

```
      Operand_1     : Float           := 0.0;
      Operand_2     : Float           := 0.0;

      use type Data.String_Access_Type;
      begin

      Clear;

      while Postfix(Current_Index) /= null loop

         Ada.Text_Io.Put(Postfix(Current_Index).all & ' ');

         if Is_Operand(Postfix(Current_Index))
            then

            Number := Float'Value(Postfix(Current_Index).all);
            Push(Number);

            else

            Operand_1 := Value;
            Pop;

            Operand_2 := Value;
            Pop;

            if Postfix(Current_Index).all = "+"
               then
               Number := Operand_2 + Operand_1;

               elsif Postfix(Current_Index).all = "-"
               then
               Number := Operand_2 - Operand_1;

               elsif Postfix(Current_Index).all = "*"
               then
               Number := Operand_2 * Operand_1;

               elsif Postfix(Current_Index).all = "/"
               then
               Number := Operand_2 / Operand_1;

            end if;

            Push(Number);

         end if;
         Current_Index := 1 + Current_Index;
      end loop;
      return Value;
   end Calcul_From_Postfix;
```

15.4.8 Le module affichage du résultat

15.4.8.1 Définition des procédure et fonctions

```
Procedure affichage.to_screen ( value : in float);
```

15.4.8.2 Traitement

IL s'agit d'afficher le résultat sur l'écran sans zéros non significatifs ni exposant ni espaces.
Par exemple:
 0.0000333333
 0.0000001137
 123456.78
 32267.7

Nous nous trouvons dans le cas du premier module, il faut utiliser une partie du langage qui ne nous est pas familière :

Voici donc comme point de départ, extrait du manuel ada, les instructions qui permettent l'impression des réels:

A.10.9 Input-Output for Real Types

Static Semantics

The following procedures are defined in the generic packages Float_IO, Fixed_IO, and Decimal_IO, which have to be instantiated for the appropriate floating point, ordinary fixed point, or decimal fixed point type respectively (indicated by Num in the specifications).

Values are output as decimal literals without low line characters. The format of each value output consists of a Fore field, a decimal point, an Aft field, and (if a nonzero Exp parameter is supplied) the letter E and an Exp field. The two possible formats thus correspond to: Fore . Aft and to: Fore . Aft E Exp without any spaces between these fields. The Fore field may include leading spaces, and a minus sign for negative values. The Aft field includes only decimal digits (possibly with trailing zeros). The Exp field includes the sign (plus or minus) and the exponent (possibly with leading zeros). For floating point types, the default lengths of these fields are defined by the following variables that are declared in the generic package Float_IO:

```
Default_Fore : Field := 2;
Default_Aft  : Field := Num'Digits-1;  *
Default_Exp  : Field := 3;
```

For ordinary or decimal fixed point types, the default lengths of these fields are defined by the following variables that are declared in the generic packages Fixed_IO and Decimal_IO, respectively:

```
Default_Fore : Field := Num'Fore;  **
 Default_Aft  : Field := Num'Aft;     ***
 Default_Exp  : Field := 0;

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

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

Outputs the value of the parameter Item as a decimal literal with the format defined by Fore, Aft and Exp. If the value is negative, or if Num is a floating point type where Num'Signed_Zeros is True and the value is a negatively signed zero, then a minus sign is included in the integer part. If Exp has the value zero, then the integer part to be output has as many digits as are needed to represent the integer part of the value of Item, overriding Fore if necessary, or consists of the digit zero if the value of Item has no integer part. If Exp has a value greater than zero, then the integer part to be output has a single digit, which is nonzero except for the value 0.0 of Item. In both cases, however, if the integer part to be output has fewer than Fore characters, including any minus sign, then leading spaces are first output to make up the difference. The number of digits of the fractional part is given by Aft, or is one if Aft equals zero. The value is rounded; a value of exactly one half in the last place is rounded away from zero. If Exp has the value zero, there is no exponent part. If Exp has a value greater than zero, then the exponent part to be output has as many digits as are needed to represent the exponent part of the value of Item (for which a single digit integer part is used), and includes an initial sign (plus or minus). If the exponent part to be output has fewer than Exp characters, including the sign, then leading zeros precede the digits, to make up the difference. For the value 0.0 of Item, the exponent has the value zero.

```
procedure Put(To   : out String;
              Item : in Num;
              Aft  : in Field := Default_Aft;
              Exp  : in Field := Default_Exp);
```

Outputs the value of the parameter Item to the given string, following the same rule as for output to a file, using a value for Fore such that the sequence of characters output exactly fills the string, including any leading spaces.

Example: (b means space)

```
package Real_IO is new Float_IO(Real); use Real_IO;
 -- default format used at instantiation, Default_Exp = 3
42
X : Real := -123.4567;  --  digits 8      (see 3.5.7)
43
Put(X);  -- default format                    "-1.2345670E+02"
 Put(X, Fore => 5, Aft => 3, Exp => 2);              -- "bbb-1.235E+2"
 Put(X, 5, 3, 0);                                 -- "b-123.457"
```

* Num'Digits renvoie le nombre de chiffres significatifs
** Num'Fore; used for fixed point floats
*** Num'Aft; used for fixed point floats

Nous allons:

1. Essayer ces instructions pour comprendre les détails de leur fonctionnement.
2. Mettre en oeuvre ces instructions pour réaliser le traitement des données.

15.4.8.2.1 Exploration des possibilités du langage

15.4.8.2.2 Utilisation de « image »

```
with Ada.Text_Io;
.............
Ada.Text_Io.Put(Float'Image(x));
.............
```

Le résultat sera toujours de la forme : -1.2345670E+02, ce qui ne convient pas.

15.4.8.2.3 Utilisation du générique « Float_Io »

L'utilisation d'un générique implique une syntaxe particulière, qui s'appelle «instantiation» en anglais(voir le cours chapitre 8). C'est l'instruction `package My_Float_Io is new Ada.Text_Io.Float_Io(Float);` dans le code ci-dessous:

```
 with Ada.Text_Io;

  procedure ......
    is
    package My_Float_Io is new Ada.Text_Io.Float_Io(Float);
   .........
    begin

    My_Float_Io.Put(Item => X,
```

```
                  Fore => 3,
                  Aft  => 2,
                  Exp  => 0);
......
```

Le résultat n'est pas bien précis car le langage modifiera les paramètres « fore » et « aft » si le nombre à imprimer est trop grand ou trop petit.

15.4.8.2.4 Utilisation du générique « Float_Io » avec une chaîne de caractères

```
with Ada.Text_Io;

 procedure X
   is
   package My_Float_Io is new Ada.Text_Io.Float_Io(Float);

   Strg : String(1 .. 20) := (others => ' ');

   Value : Float;
   begin

   My_Float_Io.Get(Value);

   My_Float_Io.Put(To   => Strg,
                   Item => Value,
                   Aft  => 10,
                   Exp  => 0);

   Ada.Text_Io.Put('>'& Strg & '<');
 end X;
```

Essayons ensemble ce petit bout de programme pour voir:

```
[dg@home simple_calc_solution]$ ./x
1
>        1.0000000000<
[dg@home simple_calc_solution]$ ./x
.123456789
>        0.1234567910<
[dg@home simple_calc_solution]$ ./x
1000000
>  1000000.0000000000<
[dg@home simple_calc_solution]$ ./x
781859.158752
>   781859.1875000000<
[dg@home simple_calc_solution]$ ./x
666666.666666
>   666666.6875000000<
123456789
>123456792.0000000000<
[daniel@localhost simple_calc_solution]$ ./x
1234567890
raised ADA.IO_EXCEPTIONS.LAYOUT_ERROR : a-tiflau.adb:218
[daniel@localhost simple_calc_solution]$
```

Ici il n'y a plus de limitation mais le résultat est mis dans une chaîne de caractères avec des espaces avant, un trop grand nombre de chiffres significatifs et des zéros non significatifs. De plus si le nombre est trop grand, il y a une erreur pendant le fonctionnement qui sera capturée par le mécanisme d' »exception ». Mais c'est là la seule solution. Il faut donc :

1. Traduire le réel en une chaîne de caractères, assez longue pour la plupart des valeurs, et prévoir un affichage scientifique si le nombre est vraiment très petit ou très grand en utilisant la gestion d'exceptions.

Exemple :
βββββββββ123.45000 (β représente un espace)

2. Éliminer les espaces au début (attention au signe -).
3. Remplacer tous les chiffres non significatifs par des zéros.
4. Éliminer les zéros non significatifs (attention au point décimal).
5. Afficher la chaîne de caractères sur l'écran.
6. Implémenter la gestion d'exception avec affichage en format scientifique.

1000000.0000000000	devient	1000000
0.1234567910	devient	0.1234567
781859.158752	devient	781859.1

15.4.8.2.5 Traduction du réel en une chaîne de caractères.

C'est l'application de ce que nous venons d'étudier :

```
F_Io.Put(To   => Strg,
         Item => Value,
         Aft  => 10,
         Exp  => 0);
```

15.4.8.2.6 Élimination des espaces au début.

Un compteur (first) est introduit. Une boucle parcourt la chaîne de caractères
et avance « first » aussi longtemps que le caractère courant est un espace.

```
for I in Strg'Range loop

          if Strg(I) = ' '
             then
             First := I;
             else
             exit;
          end if;
       end loop;
```

15.4.8.2.7 Remplacement de tous les chiffres non significatifs par des zéros.

Un compteur «L» est introduit. Une boucle parcourt la chaîne de caractères
et avance «L» quand le caractère courant est un chiffre de 1 à 9. Dés que le nombre de chiffres significatifs est dépassé (if L > Float'Digits ...) tous les chiffres de 1 à 9 sont remplacés par des 0 qui seront éliminés à l'étape suivante.

```
for K in Strg'Range loop
  if Strg(K) in '1' .. '9'
     then L := 1 + L;
     if L > Float'Digits
        then Strg(K) := '0';
     end if;
  end if;
end loop;
```

15.4.8.2.8 Élimination des zéros non significatifs.

Il faut éliminer les zéros après et le point décimal si il n'y a plus de zéros (10.000 devient 10). La position du point décimal est trouvée puis une boucle « reverse » sert à éliminer les zéros (et le point décimal si nécessaire).

```
for I in Strg'Range loop
  if Strg(I) = '.'
     then
     Decimal_Point := I;
     exit;
  end if;
end loop;

if Decimal_Point /= Strg'First
  then
  for I in reverse Decimal_Point .. Strg'Last loop
     if Strg(I) = '0'
        or else Strg(I) = '.'
        then
        Last := I - 1;
        else
        exit;
     end if;
  end loop;
end if;
```

15.4.8.2.9 Affichage de la chaîne de caractères sur l'écran.

```
Ada.Text_Io.Put("= " & Strg(First .. Last));
```

15.4.8.2.10 La gestion d'exception avec affichage en format scientifique.

Ada inclut aussi une gestion d'exception qui permet de **réagir durant l'exécution du programme** à des valeurs imprévues des données. Ici on capture toutes les erreurs (when others) et on utilise dans ce cas la notation scientifique.

```
exception
when others =>
Ada.Text_Io.Put(Float'Image(Value));
```

15.4.8.3 Le code complet

```
procedure To_Screen(Value : in     Float)
       is

       Strg          : String(1 .. 50);
       Last          : Natural := Strg'Last;
       First         : Natural := Strg'First;
       Decimal_Point : Natural := Strg'First;
       L             : Integer := 0;
       package F_Io is new Ada.Text_Io.Float_Io(Float);
       begin

       F_Io.Put(To   => Strg,
                Item => Value,
                Aft  => 10,
                Exp  => 0);

       for K in Strg'Range loop
          if Strg(K) in '1' .. '9'
             then L := 1 + L;
             if L > Float'Digits
                then Strg(K) := '0';
             end if;
          end if;
       end loop;

       for I in Strg'Range loop

          if Strg(I) = ' '
             then
             First := I;
             else
             exit;
          end if;
       end loop;

       for I in Strg'Range loop
          if Strg(I) = '.'
             then
             Decimal_Point := I;
             exit;
          end if;
       end loop;

       if Decimal_Point /= Strg'First
          then
          for I in reverse Decimal_Point .. Strg'Last loop
             if Strg(I) = '0'
                or else Strg(I) = '.'
                then
                Last := I - 1;
                else
                exit;
             end if;
          end loop;
       end if;

       Ada.Text_Io.Put("= " & Strg(First .. Last));


       exception
       when others =>
       Ada.Text_Io.Put(Float'Image(Value));

    end To_Screen;
```

16 L'ALGORITHMIQUE EN ACTION, EXEMPLE 2: LE CODAGE ET DÉCODAGE "LZW"

16.1 INTRODUCTION

Ce projet est beaucoup plus ambitieux que le précédent et il fait appel à des ressources diverses et variées. Il existe plusieurs variantes différentes. Voici l'algorithme détaillé tel que compilé d'après plusieurs sources dont l'original: T.Welsh IEEE Computer June 1984.

16.1.1 L'algorithme de compression

Cet algorithme fait beaucoup appel à la table ASCII qui établit la relation entre les lettres et le code binaire correspondant. Vous la trouverez ci-dessous avec les équivalences décimales et hexadécimales.

La table ASCII

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
000	00	NUL ^@	018	12	DC2 ^R	036	24	$	054	36	6
001	01	SOH ^A	019	13	DC3 ^S	037	25	%	055	37	7
002	02	STX ^B	020	14	DC4 ^T	038	26	&	056	38	8
003	03	ETX ^C	021	15	NAK ^U	039	27	'	057	39	9
004	04	EOT ^D	022	16	SYN ^V	040	28	(	058	3A	:
005	05	ENQ ^E	023	17	ETB ^W	041	29	)	059	3B	;
006	06	ACK ^F	024	18	CAN ^X	042	2A	*	060	3C	<
007	07	BEL ^G	025	19	EM ^Y	043	2B	+	061	3D	=
008	08	BS ^H	026	1A	SUB ^Z	044	2C	,	062	3E	>
009	09	HT ^I	027	1B	ESC ^[	045	2D	-	063	3F	?
010	0A	LF ^J	028	1C	FS ^\	046	2E	.	064	40	@
011	0B	VT ^K	029	1D	GS ^]	047	2F	/	065	41	A
012	0C	FF ^L	030	1E	RS ^^	048	30	0	066	42	B
013	0D	CR ^M	031	1F	US ^_	049	31	1	067	43	C
014	0E	SO ^N	032	20	SP	050	32	2	068	44	D
015	0F	SI ^O	033	21	!	051	33	3	069	45	E
016	10	DLE ^P	034	22	"	052	34	4	070	46	F
017	11	DC1 ^Q	035	23	#	053	35	5	071	47	G
Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
072	48	H	090	5A	Z	108	6C	l	126	7E	~
073	49	I	091	5B	[	109	6D	m	127	7F	
074	4A	J	092	5C	\	110	6E	n	128	80	~
075	4B	K	093	5D	]	111	6F	o	129	81	~
076	4C	L	094	5E	^	112	70	p	130	82	~
077	4D	M	095	5F	_	113	71	q	131	83	~
078	4E	N	096	60	`	114	72	r	132	84	~
079	4F	O	097	61	a	115	73	s	133	85	~
080	50	P	098	62	b	116	74	t	134	86	~
081	51	Q	099	63	c	117	75	u	135	87	~
082	52	R	100	64	d	118	76	v	136	88	~
083	53	S	101	65	e	119	77	w	137	89	~
084	54	T	102	66	f	120	78	x	138	8A	~
085	55	U	103	67	g	121	79	y	139	8B	~
086	56	V	104	68	h	122	7A	z	140	8C	~
087	57	W	105	69	i	123	7B	{	141	8D	~
088	58	X	106	6A	j	124	7C	|	142	8E	~
089	59	Y	107	6B	k	125	7D	}	143	8F	~
Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
144	90	-	162	A2	¢	180	B4	Ž	198	C6	Æ
145	91	-	163	A3	£	181	B5	µ	199	C7	Ç
146	92	-	164	A4	€	182	B6	¶	200	C8	È
147	93	-	165	A5	¥	183	B7	·	201	C9	É
148	94	-	166	A6	Š	184	B8	ž	202	CA	Ê
149	95	-	167	A7	§	185	B9	¹	203	CB	Ë
150	96	-	168	A8	š	186	BA	º	204	CC	Ì
151	97	-	169	A9	©	187	BB	»	205	CD	Í
152	98	-	170	AA	ª	188	BC	Œ	206	CE	Î
153	99	-	171	AB	«	189	BD	œ	207	CF	Ï
154	9A	-	170	AA	ª	188	BC	Œ	206	CE	Î
153	99	-	172	AC	¬	190	BE	Ÿ	208	D0	Ð
155	9B	-	173	AD		191	BF	¿	209	D1	Ñ
156	9C	9	174	AE	®	192	C0	À	210	D2	Ò
157	9D	9	175	AF	¯	193	C1	Á	211	D3	Ó
158	9E	9	176	B0	°	194	C2	Â	212	D4	Ô
159	9F	9	177	B1	±	195	C3	Ã	213	D5	Õ
160	A0		178	B2	²	196	C4	Ä	214	D6	Ö
161	A1	¡	179	B3	³	197	C5	Å	215	D7	×
Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
216	D8	Ø	234	EA	ê	252	FC	ü			
217	D9	Ù	235	EB	ë	253	FD	ý			
218	DA	Ú	236	EC	ì	254	FE	þ			
219	DB	Û	237	ED	í	255	FF				
220	DC	Ü	238	EE	î						
221	DD	Ý	239	EF	ï						
222	DE	Þ	240	F0	ð						
223	DF	ß	241	F1	ñ						
224	E0	à	242	F2	ò						
225	E1	á	243	F3	ó						
226	E2	â	244	F4	ô						
227	E3	ã	245	F5	õ						
228	E4	ä	246	F6	ö						
229	E5	à	247	F7	÷						
230	E6	æ	248	F8	ø						
231	E7	ç	249	F9	ù						
232	E8	è	250	FA	ú						
233	E9	é	251	FB	û						

16.1.1.1 Initialisation de l'algorithme de compression

Tous les caractères de la table ASCII sont ajoutés à une table de hashing en association avec un index entier de 0 à 255 (l'utilisation d'un record est utile). L'index associé sera codé en utilisant un nombre de bits variable de 9 à 12. Deux valeurs spéciales "clear" (256) et "eod" (257) sont réservées. L'index associé au dernier ajout dans la table de hashing est conservé pour les additions futures dans cette table de hashing. Pour l'ajout, l'index est incrémenté après l'assignation. La table de hashing implémente les opérations suivantes :

- Initialize

Initialiser la table de hashing avec un indice sur 9 bits et tous les caractères du code ASCii. après l'ajout, l''index est mis à 258. Un marqueur "clear" est ajouté en début du fichier compressé au moment de sa création.

- Add(string)
  Ajout à la table de hashing d'une chaîne de caractères de longueur variable (string) avec l'index associé, si l'index = 511, 1023 ou 2047, le nombre de bits utilisés pour stocker l'index augmente de 1. Si l'index atteint 4095, un marqueur "Eod" est ajouté dans le fichier compressé et on appelle initialize.
- Exist(string) return boolean
  Test l'existence de cette chaîne dans la table de hashing.
- Read(string) return index
  Lecture de l'index associé à une chaîne de caractères de longueur variable en utilisant la recherche dans la table de hashing.

16.1.1.2 Détails de l'algorithme de compression

Appel de initialize
Omega <- Premier caractère du fichier à compresser;
Boucle tant que tout le fichier n'est pas lu
k <- caractère suivant dans le fichier à compresser.
SI exists(omega + k)
alors
omega <- omega + k
ou
écrire read(omega) (9-12 bits) dans le fichier compressé
Appel de add (omega + k)
omega <- k
fin si
fin boucle
écrire read(omega) (9-12 bits) dans le fichier compressé

16.1.2 L'algorithme de décompression

16.1.2.1 Initialisation de l'algorithme de décompression

Tous les caractères du code ASCII sont ajoutés à une tableau de chaînes de caractères de longueur variable en association avec un index entier de 0 à 255. L'index associé sera codé en utilisant un nombre de bits variable de 9 à 12. L'index associé est conservé pour les additions futures dans cette tableau. Le tableau implémente le opérations suivantes :

- Initialize
  Initialiser la tableau avec un indice sur 9 bits et tous les caractères du code ASCii. après l'ajout, l''index est mis à 258.
- Append(string)
  Ajout d'une chaîne de caractères avec l'index associé, si l'index = 511, 1023 ou 2047, le nombre de bits utilisés pour stocker l'index augmente de 1. Pour l'ajout, l'index est incrémenté après l'assignation.
- Exist(index)
  Teste l'existence d'une chaîne pour cet index dans la tableau.
- Read_table(index)
  Lecture de la chaîne de caractères de longueur variable correspondant à cet index du tableau

16.1.2.2 Détails de l'algorithme de décompression

Appel de initialize
incode <- premiers 9 bits du fichier à décompresser.
oldcode <- incode
écrire read_table(incode) dans le fichier résultat
Boucle tant qu'il reste des entiers de 9-12 bits à lire dans le fichier à décompresser.
incode <- lecture des 9-12 bits suivants dans le fichier à décompresser.
Si incode = clear table marker
alors
Appel de initialize
Lecture de incode
oldcode <- incode
fin si
si exists(incode)
alors
append_to_table (read_table(oldcode) + first char of (read_table(incode)))
ou
append_to_table (read_table(oldcode) + first char of (read_table(oldcode)))

fin si
écrire read_table(incode)dans le fichier résultat
oldcode <- incode
fin boucle

16.1.3 L'algorithme LZW, détails supplémentaires

Ce type de codage consiste à remplacer une chaîne de caractère de longueur variable (de 1 à n) par un nombre de bits compris entre 9 et 12. C'est l'inverse du code de hamming qui remplace chaque caractère par un nombre variable de bits.

Les données codées en utilisant la méthode de compression LZW consistent en une suite de données de longueur variable entre 9 et 12 bits. Chaque code représente un seul caractère si le code est compris entre 0 et 255, un marqueur "clear table" 256, un marqueur "eod" 257, ou un chaîne de caractères déjà rencontrée auparavant si le code est supérieur à 257.

Au début la longueur est fixée à 9 bits et la table contient seulement les entrées correspondant au 258 codes définis par tous les caractères de la table ASCII et les marqueurs. Au fur et à mesure que le codage progresse, de nouvelles entrées sont ajoutées à la table. Elles associent ces nouveaux code avec des chaînes de caractères de plus en plus longues. Bien entendu, le codeur et le décodeur construisent les mêmes tables.
Au moment ou le codeur et le décodeur découvrent (de manière synchrone et indépendante) que le nombre de bits est devenu insuffisant, ils augmentent de 1 le nombre de bits utilisés. Le premier code qui à une longueur de 10 bits est celui qui suit la création de l'entrée de valeur 511, de même après 1023 (11 bits) et 2047 (12 bits). Comme les codes ne sont jamais plus longs que 12 bits, 4095 est la dernière entrée dans la table. Dans ce cas la table est ré-initialisée comme au début et on continue avec 9 bits.

En résumé:
le codeur accumule une chaîne de caractères en concaténant les caractère en entrée tant que la chaîne correspondante est déjà présente (a déjà été rencontrée).
Sinon: il envoie en sortie ce code
Il crée un nouvelle entrée dans la table avec cette nouvelle chaîne de caractères. Cette nouvelle chaîne est égale à la chaîne accumulée plus le nouveau caractère. Et ainsi de suite.

16.1.3.1 Exemple de codage

La chaîne à coder est E E E E E e E E E f en hexadécimal : 45 45 45 45 45 65 45 45 45 66 :

Suite de caractères en entrée (Hexadécimal)	**Code de sortie** (décimal)	**Indice des chaînes ajoutées à la table** (décimal)	**Chaînes de caractères correspondant à ces codes** (Hexadécimal)
Aucun, initialisation	256 (clear table)	Non applicable	Non applicable
E (45)	45	258	E E (45 45)
E E (45 45)	258	259	E E E (45 45 45)
E E (45 45)	258	260	E E e (45 45 65)
e (65)	65	261	e E(65 45)
E E E (45 45 45)	259	262	E E E f (45 45 45 66)
f (66)	66	Non applicable	Non applicable
Aucun terminaison	257 (EOD)	Non applicable	Non applicable

Dans cet exemple, le codage fonctionne comme suit:

1. Initialisation: (chargement de tous les caractères de la table ASCI) et envoi du marqueur "clear table" code de sortie 245
2. Lecture du premier caractère ici: E, la chaîne (ici E) est dans la table: continuer la lecture
3. Lecture du second caractère ici: E, la chaîne (ici EE) n'est pas dans la table
   Envoi du code pour la chaîne précédente (E) code de sortie 45
   Ajout de EE dans la table indice 258, la chaîne est maintenant égale au dernier caractère lu ici E
4. Lecture du troisième caractère ici E, la chaîne (ici EE) est dans la table:
5. Lecture du quatrième caractère ici E, la chaîne (ici EEE) n'est pas dans la table:
   Envoi du code pour la chaîne précédente (EE) code de sortie 258
   Ajout de EEE dans la table indice 259, la chaîne est maintenant égale au dernier caractère lu ici E
6. Lecture du cinquième caractère ici E, la chaîne (ici EE) est dans la table:
7. Lecture du sixième caractère ici: e, la chaîne (ici EEe) n'est pas dans la table:
   Envoi du code pour la chaîne précédente (EE) code de sortie 258

Ajout de EEe dans la table indice 260, la chaîne est maintenant égale au dernier caractère lu ici e
8. Lecture du septième caractère ici: E la chaîne (ici eE) n'est pas dans la table:
   Envoi du code pour la chaîne précédente (e) code de sortie 65
   Ajout de eE dans la table indice 261, la chaîne est maintenant égale au dernier caractère lu ici E
9. Lecture du huitième caractère ici: E, la chaîne (ici EE) est dans la table:
10. Lecture du neuvième caractère ici: E la chaîne (ici EEE) est dans la table:
11. Lecture du dixième caractère ici: f, la chaîne (ici EEEf) n'est pas dans la table:
    Envoi du code pour la chaîne précédente (EEE) code de sortie 259
    Ajout de f dans la table indice 262, la chaîne est maintenant égale au dernier caractère lu ici f
12. Fin de lecture :
    Envoi du code pour la chaîne (f) code de sortie 66
    Envoi du code du marqueur (eof) code de sortie 257

Le codes sont concaténés en une suite de bits (attention au problème little endian big endian). La concaténation se fait de gauche à droite (bits de poids faible en premier à gauche). Cette suite de bits est ensuite découpés en bytes (8 bits) pour être écrit dans le fichier de sortie. En conséquence, des codes peuvent être divisés entre deux bytes successifs. Le début du code étant dans la fin d'un byte et la fin du code dans le début du byte suivant. Après le marqueur de fin (eod) les bits inutilisés du dernier byte sont replis de 0.

Comme l'exemple précédent est court, tous les codes sont stockés sur 9 bits. Les codes seront colorés pour vous permettre de suivre le devenir des bits lors du passage 9 bits vers 8 bits (bytes). Ici, 8 codes de 9 bits deviennent 9 bytes de 8 bits.

1. Valeurs décimale des 8 codes sur 9 bits (voir le tableau ci-dessus):
   256 45 258 258 65 259 66 257
2. Valeurs binaires de ces 8 codes sur 9 bits:
   100000000 000101101 100000010 100000010 001000001 100000011 001000010 100000001
3. Les mêmes bits concaténés :
   100000000000101101100000010100000010001000001100000011001000010100000001
4. Les mêmes bits découpes par bytes (8 bits) ce sont ces valeurs qui sont stockés dans le fichier compressé:
   10000000 00001011 01100000 01010000 00100010 00001100 00001100 10000101 00000001
5. Les mêmes en hexadécimal:
   80 0B 60 50 22 0C 0C 85 01

Pour s'adapter à un changement dans l'entrée des données à encoder, le codeur peut a tout moment, envoyer un marqueur "clear table". De cette façon, le codeur et le décodeur initialiserons les tables. Bien entendu ce marqueur sera obligatoirement envoyé quand tous les codes à 12 bits seront utilisés. Ce type de codage peut être utilisé sans connaître à priori les fréquences d'apparition des caractères (par exemple pour un canal de communication).

16.1.3.2 Exemple de décodage

Soit à décoder le résultat de l'exemple de codage précédent : 80 0B 60 50 22 0C 0C 85 01 (hexadécimal)
1. Les mêmes bits en binaire découpés par bytes
   10000000 00001011 01100000 01010000 00100010 00001100 00001100 10000101 00000001
2. Les mêmes bits après concaténation
   100000000000101101100000010100000010001000001100000011001000010100000001
3. Les mêmes découpés par groupes de 9 bits
   100000000 000101101 100000010 100000010 001000001 100000011 001000010 100000001
4. Les mêmes groupes en décimal
   256 45 258 258 65 259 66 257

Les détails du décodage:

1. Lecture de 256: c'est le marqueur "clear table" Initialisation de la table de décodage.
2. Lecture de 45 : incode est à 45
   Écriture de table(54) dans le fichier décompressé soit E
3. Lecture de 258, incode est à 258
   Table (258) n'existe pas: Ajout de table(54) & le premier caractère de Table(54) soit EE à la table pour l'index 258
   Écriture de table(258) dans le fichier décompressé soit EE
   Oldcode est à 258
4. Lecture de 258, incode est à 258
   Table (258) existe: Ajout de table(258) & le premier caractère de Table(258) soit EEE à la table pour l'index 259
   Écriture de table(258) dans le fichier décompressé soit EE
   Oldcode est à 258
5. Lecture de 65, incode est à 65
   Table (65) existe: Ajout de table(258) & le premier caractère de Table(65) soit EEe à la table pour l'index 260
   Écriture de table(65) dans le fichier décompressé soit e
6. Lecture de 259, incode est à 259
   Table (259) existe: Ajout de table(65) & le premier caractère de Table(259) soit eE à la table pour l'index 261
   Écriture de table(259) dans le fichier décompressé soit EEE
   Oldcode est à 259
7. Lecture de 66, incode est à 66

Table (66) existe: Ajout de table(259) & le premier caractère de Table(66) soit EEEf à la table pour l'index 262
Écriture de table(66) dans le fichier décompressé soit F
Oldcode est à 66

8. Lecture de 257, incode est à 66: c'est le marqueur de fin eod : fin de décompression.

16.2 STRUCTURES ET OPÉRATIONS NÉCESSAIRES

Un projet de cette dimension (bien que très réduite) ne peut réussir qu'après une reflexion intellectuelle prolongée et détaillée. Chaque structure sera implémentée et testée indépendamment, le tout étant réuni à la fin.

16.2.1 Chaînes de caractères de longueur variable

Un type de données contenant une chaîne de caractères de longueur variable est nécessaire pour le codage et le décodage. Cette "structure" doit permettre l'implémentation des opérations de création, d'assignation, d'égalité, de concaténation et de lecture du premier caractère.

La déclaration s'écrit:

```
type String_Access_Type is access all String;
```

S'agissant d'un pointeur il faut toujours prévoir son effacement avec récupération de l'espace mémoire.

```
procedure Free is new Ada.Unchecked_Deallocation(Object => String,
                                                 Name   => String_Access_Type);
```

Création et assignation (x est une chaîne de caractères de longueur variable, string_data une variable de type String_Access_Type) :

```
Free(string_data);
String_Data := new String'(X);
```

Égalité:

```
string_data_1.all=string_data_2.all
```

La concatenation fait partie des opérateurs définis par le langage:

```
String_Data := new String'(X & Y);
```

Le premier caractère d'un chaîne s'obtient par:

```
x(x'first)
```

16.2.2 Table de hashing et opérations associées au codage

L'emploi d'une table de hashing à taille fixe n'est pas envisageable car toutes les combinaisons de caractères sont possibles et il n'est pas possible de réserver d'avance deux séquences de caractères. En effet, deux combinaisons de caractères impossibles à rencontrer sont nécessaires pour l'emploi d'une table de hashing à taille fixe pour indiquer une place vide et une place déjà occupée.

Une table de hashing avec pointeur sera utilisé. Celle décrite dans ce cours convient parfaitement. Il n'y a aucune difficulté particulière pour le codage des opérations nécessaires.

Il faut implémenter:

16.2.2.1 Initialisation de la table de hashing du codage

Initialiser la table de hashing avec un indice sur 9 bits et ajouter tous les caractères du code ASCII. Après l'ajout, l''index est mis à 258. Un marqueur "clear" est ajouté en début du fichier compressé.

```
procedure Initialize
   is
Indispensable pour traduire un caractère en une chaîne de 1 caractère
   subtype One_String_Type is String(1 .. 1);
   One_String : One_String_Type;
   begin
Initialisation de la table de hashing
   H_Table.Initialize(Hash_Table => My_Hash_Table);
Ajout de la table ASCII
   for Ascii_Code in 0 .. 255 loop
Traduction d'un caractère en chaîne de 1 caractère
      One_String(1) := Character'Val(Ascii_Code);

      Add(X => One_String);
```

```
   end loop;
Initialisation de l'index courant et du nombre de bits
   Current_Index          := 258;
   Current_Number_Of_Bits := 9;
Écriture dans le fichier compressé du marqueur "clear table"
   Lzw_Io_Compress.Write(Item          => Lzw_Data.Clear_Table_Marker,
                         Number_Of_Bits => Current_Number_Of_Bits);
end Initialize;
```

16.2.2.2 Ajout à la table de hashing du codage

Ajout d'une chaîne de caractères de longueur variable avec l'index associé, si l'index = 511, 1023 ou 2047, le nombre de bits utilisés pour stocker l'index augmente de 1. Si l'index atteint 4095, un marqueur "Eod" est ajouté en début du fichier compressé et on appelle initialize.

```
procedure Add(X : in     String)
is
use type Lzw_Data.Modulo_16_Type;

Data : My_Data_Type;
begin
Donnée à stocker : indice + chaine
Data := (String_Data => new String'(X),
         Index       => Current_Index);
Ajout à la table de hashing
H_Table.Add(Data            => Data,
            To_The_Hash_Table => My_Hash_Table,
            Bucket_Number     => H_Table.H_Of(The_String => X));
Test si il faut modifier le nombre de bits
if Current_Index = 511 or else Current_Index = 1023 or else Current_Index = 2047
   then
   Current_Number_Of_Bits := 1 + Current_Number_Of_Bits;
 Test si il faut  réinitialiser
   elsif Current_Index = 4095
   then
Appel du code initialisation
   initialize;
end if;
Indice incrémenté APRÈS l'assignation
Current_Index := 1 + Current_Index;

end Add;
```

16.2.2.3 Existe dans la table de hashing du codage

Test l'existence de cette chaîne dans la table de hashing.

```
function Exists(X : in     String)
return Boolean
is
Bool    : Boolean;
My_Data : My_Data_Type;
begin
Créer les données pour la comparaison
My_Data := (String_Data => new String'(X),
            Index       => 0);--unused: the string is used as the key
Appel de l'opération correspondante de la table de hashing
H_Table.Seek(Data              => My_Data,
             In_The_Hash_Table => My_Hash_Table,
             Success           => Bool,
             Found_Record      => My_Data,
             Bucket_Number     => H_Table.H_Of(The_String => X));

return Bool;
end Exists;
```

16.2.2.4 Lecture de la table de hashing du codage

Lecture de l'index associé à une chaîne de caractères de longueur variable.

```
function Read(From : in     String)
return Lzw_Data.Modulo_16_Type
is
Bool    : Boolean;
My_Data : My_Data_Type;
begin
Créer les données pour la recherche
My_Data := (String_Data => new String'(From),
            Index       => 0);--unused: the string is used as the key
Appel de l'opération recherche de la table de hashing
H_Table.Seek(Data=> My_Data,
      In_The_Hash_Table => My_Hash_Table,
```

```
      Success    => Bool,
      Found_Record      => My_Data,
      Bucket_Number     => H_Table.H_Of(The_String => From));

return My_Data.Index;
end Read;
```

16.2.3 Tableau de chaînes de caractères de longueur variable et opérations associées au décodage

Un tableau de pointeur vers une chaîne de caractères permettra l'implémentation des opérations souhaitées. Une case du tableau n'est occupée que si le pointeur correspondant n'est pas null. Le tableau est déclaré comme suit:

```
type Modulo_16_Type is mod 2 ** 16;
for Modulo_16_Type'Size use 16;
........
type Table_Type is array (Modulo_16_Type range 0 .. 2 ** 12 - 1) of String_Access_Type;
Table : Table_Type := (others => null);
```

16.2.3.1 Initialisation du tableau de chaînes de caractères du décodage

Initialiser la tableau avec un indice sur 9 bits et tous les caractères du code ASCii. après l'ajout, l''index est mis à 258.

```
procedure Initialize
   is
Indispensable pour traduire un caractère en une chaîne de 1 caractère
   subtype One_String_Type is String(1 .. 1);
   One_String : One_String_Type;
   begin
Suppression des pointeurs et récupération de l'espace mémoire correspondant
   for I in Table'Range loop
      Free(Table(I));
   end loop;
Ajout de la table ASCII
   for Ascii_Code in 0 .. 255 loop
      One_String(1) := Character'Val(Ascii_Code);
      Append_To_Table(X => One_String);
   end loop;
Initialisation de l'index et du nombre de bits
   Current_Index         := 258;
   Current_Number_Of_Bits := 9;

end Initialize;
```

16.2.3.2 Ajout au tableau de chaînes de caractères du décodage

Ajout d'une chaîne de caractères avec l'index associé, si l'index = 511, 1023 ou 2047, le nombre de bits utilisés pour stocker l'index augmente de 1. Pour l'ajout, l'index est incrémenté après l'assignation.

```
procedure Append_To_Table(X : in     String)
   is
   begin
Donnée à stocker :   chaîne de caractères
   Table(Current_Index) := new String'(X);
Indice incrémenté APRÈS l'assignation
   Current_Index        := 1 + Current_Index;
Si l'index = 511, 1023 ou 2047, le nombre de bits utilisés pour stocker l'index augmente de 1
   if Current_Index = 511 or else Current_Index = 1023 or else Current_Index = 2047
      then
      Current_Number_Of_Bits := 1 + Current_Number_Of_Bits;
   end if;
end Append_To_Table;
```

16.2.3.3 Existe dans le tableau de chaînes de caractères du décodage

Teste l'existence d'une chaîne quelle qu'elle soit pour cet indice dans la tableau.

```
function Exists(Index : in     Lzw_Data.Modulo_16_Type)
   return Boolean
   is
   begin
Si le contenu (pointeur vers une chaîne de caractères) n'est pas null, une chaîne quelle qu'elle soit existe
   return Table(Index) /= null;
end Exists;
```

16.2.3.4 Lecture du tableau de chaînes de caractères du décodage

Renvoie la chaîne de caractères correspondant à un indice.

```
function Read_Table(Index : in    Modulo_16_Type)
   return String
   is
   begin
Renvoie la chaîne de caractères correspondant à un indice (noter la dé-référence)
   return Table(Index).all;
end Read_Table;
```

16.2.4 Gestion de variables, codées sur 8, 9, 10, 11 ou 12 bits sans signe

Ce problème est résolu le plus simplement en utilisant une variable de type modulo et en conservant la taille en bits dans une variable globale. Le même type sera utilisé comme indice du tableau pour le décodage car il y est utilisé directement. Un autre avantage es type modulo est l'accès aux opérateurs logiques sur les bits pour l'utilisation de masques ainsi que la possibilité de faire des décalages par division ou multiplication par des puissances de 2. Ces opérations seront nécessaires pour traduire l'arrivée de bits groupés par 9, 10, 11 ou 12 bits et leur écriture par groupe de 8 (codage) et inversement (décodage). Toutes ces définitions seront déclarées dans un package spécial qui sera utilisé par tous les modules qui en ont besoin.

```
package Lzw_Data
is
........
type Modulo_16_Type is mod 2 ** 16;
for Modulo_16_Type'Size use 16;
.....
```

16.2.5 Lecture d'un fichier par bytes, envoi au décodage et écriture codée sur 8, 9, 10, 11 ou 12 bits sans signe dans un fichier

C'est probablement la partie la plus délicate à coder. Il faut ici dissocier absolument la lecture du fichier de la traduction et de l'écriture des codes. Une interface simple, du type itérateur devra être proposé à la routine de codage pour conserver sa simplicité d'écriture.

16.2.5.1 Lecture des caractères depuis le fichier à coder et mise à disposition pour la routine de codage.

L'interface avec la routine de codage peut se concevoir comme un "itérateur" muni des opérations suivantes:

1. **Initialiser la lecture**: Ouverture du fichier sur disque et initialisation de l'itérateur.
2. **Lecture terminée**: L'itérateur a visité dans l'ordre tous les caractères du fichier.
3. **Caractère courant**: Renvoi le caractère courant.
4. **Passage au suivant**: Avance au caractère suivant du fichier.

Initialiser la lecture

Ouverture du fichier sur disque et initialisation de l'itérateur
Pour accélérer le fonctionnement le code utilise un tampon. Pour homogénéiser le code avec l'écriture nous utiliseront "stream io" qui permet de lire et d'écrire des nombre de bits variable.

```
function Init_Reading(File_Name : in     String)
   return Boolean
   is
   begin
Fermer le fichier si déjà ouvert (c'est une bonne précaution)
   if Ada.Streams.Stream_Io.Is_Open(File => The_File_In)
      then
      Ada.Streams.Stream_Io.Close(File => The_File_In);
   end if;
Initialiser les drapeaux utilisés
   Current_Character := Ascii.Nul;
   Done              := False;
```

```
   Read_Count        := 1;
Ouverture du fichier ???
   Ada.Streams.Stream_Io.Open(File => The_File_In,
                              Name => File_Name,
                              Mode => Ada.Streams.Stream_Io.In_File);
Si ok lire un block depuis le fichier et initialiser le premier caractère du fichier pour que caractère courant soit prêt
   Get_Next_Block;
   Next_Character;
Renvoi ok pour cette opération
   return True;
Fichier n'existe pas?
   exception
   when others =>
   Done := True;
   Ada.Streams.Stream_Io.Close(File => The_File_In);
    return False;
end Init_Reading;
```

Nous avons besoin de deux autre routines Get_Next_Block et Next_Character décrit plus bas

```
procedure Get_Next_Block
   is
Cette fonction traduit un block de bytes lu du fichier texte en un block de caractéres
   function Data_To_Block is new Unchecked_Conversion(Source => Read_Stream_Data_Type,
                                                      Target => Lzw_Data.Block_Type);
   begin
Lire un block depuis le fichier à coder:
Si Read_Length =0 le fichier est lu en entier et sa longueur est un multiple de 64
Si Read_Length < 64 le fichier est lu en entier et il reste length à traiter
Si Read_Length = 64 la lecture du fichier n'est pas terminée et le block est plein
   Ada.Streams.Stream_Io.Read(File => The_File_In,
                              Item => Read_Data,
                              Last => Read_Length,
                              From => Read_Count);
Traduction bytes en caractères
   Read_Block := Data_To_Block(Read_Data);
Conserver l'endroit ou la lecture se trouve dans le fichier pour la prochaine opération de lecture
   Read_Count := Ada.Streams.Stream_Io.Count(128) + Read_Count;
Fini=> mettre le drapeau
Last_Block_In_File_Read := Integer(Read_Length) < 128 ;-- done
end Get_Next_Block;
```

Lecture terminée

L'itérateur a visité dans l'ordre tous les caractères du fichier: Renvoie le drapeau correspondant. La simplicité du code ne doit pas faire illusion: Ces routines doivent être séparés du programme de codage pour respecter la division des tâches, permettre une lecture facile et éliminer les accès par le programme de codage aux détails et variables utilisés par la lecture qui ne sont pas nécessaire au strict fonctionnement.

```
function Reading_Over
   return Boolean
   is
   begin
   return Done;
end Reading_Over;
```

Caractère courant

Renvoi le caractère courant: La simplicité du code ne doit pas faire illusion: Ces routines doivent être séparés du programme de codage pour respecter la division des tâches, permettre une lecture facile et éliminer les accès par le programme de codage aux détails et variables utilisés par la lecture qui ne sont pas nécessaire au strict fonctionnement.

```
function Value
   return Character
   is
   begin
   return Current_Character;
end Value;
```

Passage au suivant

Avance au caractère suivant du fichier.

```
procedure Next_Character
   is
   begin
   Done := Last_Block_In_File_Read and Current_Position_In_Block = 1 + Integer(Read_Length);
Si la lecture n'est pas terminée vérifier que tous les caractères ont été utilisés
   if not Done
      then
Non: passer au caractère suivant dans le block
      Current_Character         := Read_Block(Current_Position_In_Block);
```

```
      Current_Position_In_Block := 1 + Current_Position_In_Block;
C'était le dernier caractère du block ? Si oui lire le block suivant
       if Current_Position_In_Block > 128
         then
         Current_Position_In_Block := 1;
         Get_Next_Block;
      end if;
   end if;
end Next_Character;
```

16.2.5.2 Réception des codes depuis la routine de codage, traduction en bytes et écriture dans un fichier.

L'interface avec la routine de codage sera implémentée avec:

1. **Initialisation** :
   Ouverture du fichier en écriture
2. **Écrire** :
   Réception des 9, 10, 11 ou 12 bits et écriture disque par bytes
   Fermeture:
3. Vider le tampon des bits et fermeture du fichier disque

Initialisation

Ouverture du fichier en écriture

```
function Init_Writing(File_Name : in     String)
   return Boolean
   is
   begin
Fermer le fichier si déjà ouvert (c'est une bonne précaution)
   if Ada.Streams.Stream_Io.Is_Open(File => The_File_Out)
      then
      Ada.Streams.Stream_Io.Close(File => The_File_Out);
   end if;
Ouverture du fichier ???
   Ada.Streams.Stream_Io.Create(File => The_File_Out,
                               Name => File_Name,
                               Mode => Ada.Streams.Stream_Io.Out_File);
Traduction de  The_File_Out en   File_Stream_Out (nécessaire pour écriture)
   File_Stream_Out := Ada.Streams.Stream_Io.Stream(File => The_File_Out);
Renvoi succès
   return True;
Erreur ?
   exception
   when others =>
   Done := True;
   return False;
end Init_Writing;
```

Écrire

Réception des 9, 10, 11 ou 12 bits et écriture disque par bytes

```
procedure Write(Item          : Lzw_Data.Modulo_16_Type;
                Number_Of_Bits : Lzw_Data.Number_Of_Bits_Type)
   is
Nécessaire en ada pour avoir les opérateurs associés
   use type Lzw_Data.Modulo_8_Type;
   use type Lzw_Data.Modulo_16_Type;
   use type Lzw_Data.Modulo_32_Type;
Définition du tampon de stockage et du byte à écrire
   Byte_To_Write_On_File : Lzw_Data.Modulo_8_Type;
   Temporary_Storage     : Lzw_Data.Modulo_32_Type;
   begin
Mettre les bits reçus depuis une variable sur 16 bits dans une variable sur 32 bits (ces bits sont alignés à droite)
   Temporary_Storage := Lzw_Data.Modulo_32_Type(Item);
Mettre ces bits à la bonne position à gauche en décalant
   Temporary_Storage := Temporary_Storage * 2 ** (Free_Position - Number_Of_Bits + 1);
Introduction de ces bits dans le tampon à leur place avec l'opérateur OU
   Modulo_32 := Modulo_32 or Temporary_Storage;
Calcul de la nouvelle valeur du premier bit libre pour le prochain APPEL
   Free_Position := Free_Position - Number_Of_Bits;
Maintenant il faut écrire le ou les bytes disponibles dans le fichier ( en effet si il reste 7 bits et que l'on en ajoute 11 il faut écrire 2 fois 8 bits et il en restera 2)
   while Free_Position < 23 loop
```

```
Mettre les 8 bits qui sont le plus à gauche dans une variable temporaire
      Temporary_Storage := Modulo_32 / 2 ** 24;
Mettre les 8 bits depuis une variable sur 32 bits dans une variable sur 8 bits
      Byte_To_Write_On_File := Lzw_Data.Modulo_8_Type(Temporary_Storage);
écriture disque
      Lzw_Data.Modulo_8_Type'Write(File_Stream_Out,
                                   Byte_To_Write_On_File);
Réajustement du premier bit libre dans le tampon
      Free_Position := Free_Position + 8;
Effacement des bits écrits sur disque
      Modulo_32 := Modulo_32 * 2 ** 8;
   end loop;
end Write;
```

Fermeture

Vider le tampon des bits et fermeture du fichier disque

Le mécanisme décrit pour l'écriture disque implique qu'il peut rester des bits non sauvés dans le fichier. Cette routine sera appelée par la routine de codage pour vider le tampon et fermer le fichier

```
procedure Flush_Buffer_And_Close_File
   is
   begin
Envoyer le marqueur eod cela forcera une écriture disque des bits restant dans le tampon
   Write(Item          => Lzw_Data.Eod_Marker,
         Number_Of_Bits => 9);
Fermeture des fichiers ouverts
   Ada.Streams.Stream_Io.Close(File => The_File_Out);
   Ada.Streams.Stream_Io.Close(File => The_File_In);
end Flush_Buffer_And_Close_File;
```

16.2.6 Réception de variables codées sur 8, 9, 10, 11 ou 12 bits sans signe, envoi au décodage et écriture dans un fichier par bytes

Les problèmes sont du même ordre que pour le codage et seront résolus d'une manière similaire.

16.2.6.1 Lecture des codes depuis le fichier à décoder et mise à disposition pour la routine de décodage.

L'interface avec la routine de décodage peut se concevoir comme un "itérateur" muni des opérations suivantes:

1. **Initialiser la lecture**: Ouverture du fichier sur disque et initialisation de l'itérateur.
2. **Lecture terminée**: L'itérateur a visité dans l'ordre tous les codes du fichier.
3. **Code courant**: Renvoi le code courant.
4. **Passage au suivant**: Avance au code suivant dans le fichier.

Initialiser la lecture

Ouverture du fichier sur disque et initialisation de l'itérateur.

```
function Init_Reading(File_Name      : in     String;
                      Number_Of_Bits : in     Lzw_Data.Number_Of_Bits_Type)
   return Boolean
   is
   begin
Fermer le fichier si déjà ouvert (c'est une bonne précaution)
   if Ada.Streams.Stream_Io.Is_Open(File => The_File_In)
      then
      Ada.Streams.Stream_Io.Close(File => The_File_In);
   end if;
Ouverture si le fichier existe sinon exception
   Ada.Streams.Stream_Io.Open(File => The_File_In,
                              Name => File_Name,
                              Mode => Ada.Streams.Stream_Io.In_File);
Traduction de  The_File_In en   File_Stream_In (nécessaire pour lecture)
   File_Stream_In := Ada.Streams.Stream_Io.Stream(File => The_File_In);
Initialisation avec le chargement du premier code
  Next_Item(Number_Of_Bits => Number_Of_Bits);
Ok ?
   return True;
Fichier non trouvé?
   exception
   when others =>
   Done := True;
```

```
   return False;
end Init_Reading;
```

Lecture terminée

L'itérateur a visité dans l'ordre tous les codes du fichier.

```
function Reading_Over
   return Boolean
   is
   begin
   return Done;
end Reading_Over;
```

Code courant

Renvoi le code courant.

```
function Value
   return Lzw_Data.Modulo_16_Type
   is
   begin
   return Item;
end Value;
```

Passage au suivant

Avance au code suivant dans le fichier. La lecture se fait comme l'écriture par 8 bits.

```
procedure Next_Item(Number_Of_Bits : in     Lzw_Data.Number_Of_Bits_Type)
   is
Deux variables: une de 8 bits et une de 32 bits pour le stockage et les décalages
   Byte_Read_From_File : Lzw_Data.Modulo_8_Type;
   Temporary_Storage   : Lzw_Data.Modulo_32_Type;
   begin
Lecture de suffisamment de bits pour en avoir assez pour fournir le nombre de bits demandés
   while Free_Position > 31 - Number_Of_Bits loop
Lire 8 bits
      Lzw_Data.Modulo_8_Type'Read(File_Stream_In,
                                  Byte_Read_From_File);
Traduction d'une variable sur 8 bits à une variable sur 32 bits
      Temporary_Storage := Lzw_Data.Modulo_32_Type(Byte_Read_From_File);
Décaler ces 8 bits a gauche jusqu'à la position libre
      Temporary_Storage := Temporary_Storage * 2 ** (Free_Position - 7);
Introduction de ces bits dans le tampon à leur place avec l'opérateur OU
      Modulo_32 := Modulo_32 or Temporary_Storage;
Calcul de la nouvelle valeur du premier bit libre pour le prochain tour de boucle
      Free_Position := Free_Position - 8;
   end loop;
Décalage à droite pour conserver le nombre de bits demandés par la routine de décodage
   Temporary_Storage := Modulo_32 / 2 ** (32 - Number_Of_Bits);
Traduction d'une variable sur 32 bits à une variable sur 16 bits
   Item := Lzw_Data.Modulo_16_Type(Temporary_Storage);
Calcul de la nouvelle valeur du premier bit libre pour le prochain APPEL
   Free_Position := Free_Position + Number_Of_Bits;
Effacement des bits envoyés au décodage
   Modulo_32 := Modulo_32 * 2 ** Number_Of_Bits;
Gestion d'erreur
   exception
   when Ada.Io_Exceptions.End_Error =>
   Done := True;
   Ada.Streams.Stream_Io.Close(File => The_File_In);
end Next_Item;
```

16.2.6.2 Réception des caractères depuis la routine de décodage et écriture dans un fichier.

L'interface avec la routine de décodage sera implémentée avec:

1. **Initialisation** :
   Ouverture du fichier en écritu
2. **Écrire** :
   écriture disque par caractères
3. **Fermeture**:
   Fermeture du fichier disque

Initialisation

Ouverture du fichier en écriture

```
function Init_Writing(File_Name : in     String)
   return Boolean
   is
   begin
Fermer le fichier si déjà ouvert (c'est une bonne précaution)
```

```
   if Ada.Streams.Stream_Io.Is_Open(File => The_File_Out)
      then
      Ada.Streams.Stream_Io.Close(File => The_File_Out);
   end if;
Ouverture si le fichier existe sinon exception
   Ada.Streams.Stream_Io.Create(File => The_File_Out,
                                Name => File_Name,
                                Mode => Ada.Streams.Stream_Io.Out_File);
Traduction de The_File_Out en   File_Stream_Out (nécessaire pour l'écriture)
   File_Stream_Out := Ada.Streams.Stream_Io.Stream(File => The_File_Out);
Ok
   return True;
Problème ?
   exception
   when others =>
   Done := True;
   return False;
end Init_Writing;
```

Écrire

Écriture disque par caractères

```
procedure Write(Item : in     Character)
   is
   begin
écriture après traduction en byte
   Lzw_Data.Modulo_8_Type'Write(File_Stream_Out,
                                Character'Pos(Item));
end Write;
```

Fermeture

Fermeture du fichier disque

```
procedure Close_Write
   is
   begin
Fermeture du fichier
   Ada.Streams.Stream_Io.Close(File => The_File_Out);
end Close_Write;
```

16.2.7 Le codage

Toutes les routines nécessaires sont maintenant disponibles, nous pouvons (enfin) écrire le code:

```
procedure Compress(The_File      : in     String;
                   Into_The_File : in     String)
   is

   subtype One_String_Type is String(1 .. 1);
   Bool                    : Boolean;
   Omega                   : String(1 .. 10000)       := (others => ' ');
   K                       : One_String_Type          := (others => ' ');
   Curent_Position_In_Omega : Integer range 1 .. 10000 := 1;
   -- Codes are never longer than
   -- 12 bits; therefore, entry 4095 is the last entry of the LZW table.
   Item : Lzw_Data.Modulo_16_Type range 0 .. 4095;
   use type Lzw_Data.Modulo_16_Type;
   begin
Ouverture des fichiers et appel de initialize (test bool à écrire)
   Bool := Lzw_Io_Compress.Init_Reading(File_Name          => The_File);
   Bool := Bool and Lzw_Io_Compress.Init_Writing(File_Name => Into_The_File);
   Initialize;
Omega <- Premier caractère du fichier à compresser;
   Omega(1) := Lzw_Io_Compress.Value;
   Lzw_Io_Compress.Next_Character;
Boucle tant que tout le fichier n'est pas lu
   while not Lzw_Io_Compress.Reading_Over loop
k <- caractère suivant dans le fichier à compresser.
      K(1) := Lzw_Io_Compress.Value;
Sl exists( omega + k)
alors
omega <- omega + k
      if Exists(Omega(1 .. Curent_Position_In_Omega) & K)
         then
         Curent_Position_In_Omega        := 1 + Curent_Position_In_Omega;
         Omega(Curent_Position_In_Omega) := K(1);
ou
         else
écrire read(omega)  (9-12 bits) dans le fichier compressé
         Item := Read(Omega(1 .. Curent_Position_In_Omega));

         Lzw_Io_Compress.Write(Item           => Item,
                               Number_Of_Bits => Current_Number_Of_Bits);
Appel  de add ( omega + k)
         Add(Omega(1 .. Curent_Position_In_Omega) & K);
omega <- k
```

```
         Omega(1 .. 1)            := K;
         Curent_Position_In_Omega := 1;
fin si
      end if;
Valeur lue: passer l'itérateur au suivant
      Lzw_Io_Compress.Next_Character;
fin boucle
   end loop;
écrire read(omega)  (9-12 bits) dans le fichier compressé
   Item := Read(Omega(1 .. Curent_Position_In_Omega));

   Lzw_Io_Compress.Write(Item           => Item,
                         Number_Of_Bits => Current_Number_Of_Bits);
Fermeture des fichiers
   Lzw_Io_Compress.Flush_Buffer_And_Close_File;
end Compress;
```

16.2.8 Le décodage

Toutes les routines nécessaires sont maintenant disponibles, nous pouvons (enfin) écrire le code:

```
procedure Expand(The_File      : in     String;
                 Into_The_File : in     String)
   is
Permet d'écrire une chaîne de caractères un caractère à la fois
   procedure Output(S : in     String)
      is
      begin
      for I in S'Range loop
         Lzw_Io_Expand.Write(Item => S(I));
      end loop;
   end Output;
   Bool : Boolean;
12 bits maximum (4095)
   Incode  : Lzw_Data.Modulo_16_Type range 0 .. 4095;
   Oldcode : Lzw_Data.Modulo_16_Type range 0 .. 4095;
   use type Lzw_Data.Modulo_16_Type;
   begin
Ouverture des fichiers et initialisation de l'itérateur (bool devrait être testé)
   Bool := Lzw_Io_Expand.Init_Reading(File_Name      => The_File,
                                      Number_Of_Bits => 9);

   Bool := Bool and Lzw_Io_Expand.Init_Writing(File_Name => Into_The_File);
Appel de initialize
   Initialize;
Lire le masque clear table (initialisation déjà faite)
   Lzw_Io_Expand.Next_Item(Number_Of_Bits => 9);
incode <- premiers 9 bits du fichier à décompresser.
   Incode := Lzw_Io_Expand.Value;
Valeur lue: passer l'itérateur au suivant
   Lzw_Io_Expand.Next_Item(Number_Of_Bits => 9);
oldcode <- incode
   Oldcode := Incode;
écrire read_table(incode) dans le fichier résultat
   Output(S => Read_Table(Incode));
Boucle tant qu'il reste des entiers de 9-12 bits à lire dans le fichier à décompresser.
   while not Lzw_Io_Expand.Reading_Over loop
incode <-  lecture des 9-12 bits suivants dans le fichier à décompresser.
      Incode := Lzw_Io_Expand.Value;
Lecture terminée sortie
      exit when Incode = Lzw_Data.Eod_Marker;
Si incode = clear table marker
alors
Appel de initialize
Lecture de incode
oldcode <- incode
fin si
      if Incode = Lzw_Data.Clear_Table_Marker
         then
         Current_Index          := 258;
         Current_Number_Of_Bits := 9;
         for I in 258 .. Table'Last loop
            Free(Table(I));
         end loop;
         Lzw_Io_Expand.Next_Item(Number_Of_Bits => Current_Number_Of_Bits);
         Incode  := Lzw_Io_Expand.Value;
         Oldcode := Incode;
      end if;
si exists(incode)
      if Exists(Incode)
         then
append_to_table ( read_table(oldcode) + first char of (read_table(incode)) )
         Append_To_Table(Read_Table(Oldcode)
                         &
```

```
                        Read_Table(Incode) (Read_Table(Incode)'First));
ou alors
         else
append_to_table ( read_table(oldcode) + first char of (read_table(oldcode)) )
         Append_To_Table(Read_Table(Oldcode)
                        &
                        Read_Table(Oldcode) (Read_Table(Oldcode)'First));
fin si
      end if;
écrire  read_table(incode)dans le fichier résultat
     Output(S => Read_Table(Incode));
oldcode <- incode
      Oldcode := Incode;
fin boucle
      Lzw_Io_Expand.Next_Item(Number_Of_Bits => Current_Number_Of_Bits);
   end loop;
Fermeture des fichiers
   Lzw_Io_Expand.Close_Write;
end Expand;
```

16.3 LE CODE COMPLET

16.3.1 Fichier Lzw_1.ads

```
with Linked_Hash;
with Lzw_Data;
package Lzw_1
   is
   --|
   --|                          |==================================|
   --|                          |              Compress            |
   --|                          |==================================|
   --|
   procedure Compress(The_File      : in     String;
                      Into_The_File : in     String);


   private
   Current_Number_Of_Bits : Lzw_Data.Number_Of_Bits_Type := Lzw_Data.Number_Of_Bits_Type'First;
   Current_Index          : Lzw_Data.Modulo_16_Type      := 0;
   --|
   --| index definition for the hash table should be a prime
   --|
   --|Codes are never longer than
   --| 12 bits; therefore, entry 4095 is the last entry of the LZW table.
   --| but the size of an hash table must be larger than the expected number
   --| of items because the speed of access depends heavily on the load
   --|
   --|
   --|  data as a pointer to a string and an integer
   --|
   type String_Data_Type is access all String;
   type My_Data_Type is
      record
      String_Data : String_Data_Type        := null;
      Index       : Lzw_Data.Modulo_16_Type := 0;
   end record;
   --|
   --|beware :Left may be different from right while Left.all=right.all
   --| define = for the hash table algo
   --|
   function "=" (Left  : in My_Data_Type;
                 Right : in     My_Data_Type)
   return Boolean;
   --|
   --| instantiation of hash
   --|
   package H_Table is new Linked_Hash(Data_Type => My_Data_Type,
                                      "="       => "=");

   My_Hash_Table : H_Table.Linked_Hash_Table_Type(Number_Of_Buckets => 2 ** 10 - 1);

   --My_Iterator   : H_Table.Iterator_Type;-- debug only
end Lzw_1;
```

16.3.2 Fichier Lzw_1.adb

```
with Lzw_Io_Compress;
with Lzw_Data;
with Ada.Text_Io;
--|
--|
--|                    ____________________________________
--|                   ||                                  ||
--|                   ||                                  ||
```

```
--|                  ||            Lzw_1             ||
--|                  ||                              ||
--|                  ||______________________________||
--|
--|
package body Lzw_1
  is
  --|
  --|                        |================================|
  --|                        |             =                  |
  --|                        |================================|
  --|
  --|                  N E E D E D   B Y   H A S H   T A B L E
  --|
  --| beware :left.String_Data may be different from right.String_Data
  --|  while Left.String_Data.all=right.String_Data.all
  --| define = for the hash table algo
  --|
  function "=" (Left  : in My_Data_Type;
                Right : in     My_Data_Type)
     return Boolean
     is
     begin
     return Left.String_Data.all = Right.String_Data.all;
  end "=";
  --|
  --|                        |================================|
  --|                        |              Add               |
  --|                        |================================|
  --|
  procedure Add(X : in     String)
     is
     use type Lzw_Data.Modulo_16_Type;
     -- add ( string)  storing the string and the associated incremented index
     Data : My_Data_Type;
     begin
     --|
     --| create data to be stored
     --|
     Data := (String_Data => new String'(X),
              Index       => Current_Index);
     --|
     --|  add to hash
     --|
     H_Table.Add(Data              => Data,
                 To_The_Hash_Table => My_Hash_Table,
                 Bucket_Number     => H_Table.H_Of(The_String => X));
     -- The first
     -- output code that is 10 bits long is the one following the creation of table entry
     -- 511, and similarly for 11 (1023) and 12 (2047) bits. Codes are never longer than
     -- 12 bits; therefore, entry 4095 is the last entry of the LZW table.
     --|
     --|  if true increment current number of bits
     --|
     if Current_Index = 511 or else Current_Index = 1023 or else Current_Index = 2047
        then
        Current_Number_Of_Bits := 1 + Current_Number_Of_Bits;
        --|
        --|  if true: clear dictionnary and reset
        --|
        elsif Current_Index = 4095
        then
        --|
        --|  declare data
        --|
        Clear_Dictionnary_And_Reset : declare
        subtype One_String_Type is String(1 .. 1);
        One_String : One_String_Type;
        begin
           --|
           --|  init table data structures
           --|
           H_Table.Initialize(Hash_Table => My_Hash_Table);
           --|
           --| add as strings the characters from ascii code 0 to ascii code 255
           --|
           Current_Index := 0;
           for Ascii_Code in 0 .. 255 loop
              One_String(1) := Character'Val(Ascii_Code);

              Add(X => One_String);-- recursion
           end loop;
           --|
           --|  put clear table marker
           --|
           Lzw_Io_Compress.Write(Item          => Lzw_Data.Clear_Table_Marker,
                                 Number_Of_Bits => Current_Number_Of_Bits);
           --|
           --| reset
           --|
           Current_Index          := 258;
           Current_Number_Of_Bits := 9;
        end Clear_Dictionnary_And_Reset;
     end if;
     --|
     --| increment
     --|
```

```
      Current_Index := 1 + Current_Index;

   end Add;
   --|
   --|                          |==================================|
   --|                          |            Initialize            |
   --|                          |==================================|
   --|
   procedure Initialize
      is
      -- INITIALISATION
      --
      -- add as strings the characters from ascii code 0 to ascii code 255
      -- to an hash table storing at the same time an associated index from 0 to 255
      -- the number of bits to store the index is unsigned 16 bits of which
      -- 9 to 12 are in use
      -- the last used index number will be kept for future additions to the
      -- hash table.
      subtype One_String_Type is String(1 .. 1);
      One_String : One_String_Type;
      begin
      --|
      --|  init table data structures
      --|
      H_Table.Initialize(Hash_Table => My_Hash_Table);
      --|
      --| add as strings the characters from ascii code 0 to ascii code 255
      --|
      for Ascii_Code in 0 .. 255 loop
         One_String(1) := Character'Val(Ascii_Code);

         Add(X => One_String);
      end loop;
      --|
      --| reset
      --|
      Current_Index          := 258;
      Current_Number_Of_Bits := 9;
      --|
      --| put clear mark
      --|
      Lzw_Io_Compress.Write(Item          => Lzw_Data.Clear_Table_Marker,
                            Number_Of_Bits => Current_Number_Of_Bits);
   end Initialize;
   --|
   --|                          |==================================|
   --|                          |            Exists                |
   --|                          |==================================|
   --|
   function Exists(X : in     String)
      return Boolean
      is
      Bool    : Boolean;
      My_Data : My_Data_Type;
      -- exists( string) return boolean
      begin
      --|
      --| create data
      --|
      My_Data := (String_Data => new String'(X),
                  Index       => 0);--unused: the string is used as the key
      --|
      --| test
      --|
      H_Table.Seek(Data              => My_Data,
                   In_The_Hash_Table => My_Hash_Table,
                   Success           => Bool,
                   Found_Record      => My_Data,
                   Bucket_Number     => H_Table.H_Of(The_String => X));

      return Bool;
   end Exists;
   --|
   --|                          |==================================|
   --|                          |              Read                |
   --|                          |==================================|
   --|
   function Read(From : in     String)
      return Lzw_Data.Modulo_16_Type
      is
      -- read(from string) returning the index associated to the string
      Bool    : Boolean;
      My_Data : My_Data_Type;
      begin
      --|
      --| create data
      --|
      My_Data := (String_Data => new String'(From),
                  Index       => 0);--unused: the string is used as the key
      --|
      --| read based on string as tag returning the data in found_record
      --|
      H_Table.Seek(Data              => My_Data,
                   In_The_Hash_Table => My_Hash_Table,
                   Success           => Bool,
                   Found_Record      => My_Data,
                   Bucket_Number     => H_Table.H_Of(The_String => From));
```

```
   --|
   --| return index
   --|
   return My_Data.Index;
end Read;
--|
--|                          |==================================|
--|                          |              Compress            |
--|                          |==================================|
--|
procedure Compress(The_File      : in     String;
                   Into_The_File : in     String)
   is

   subtype One_String_Type is String(1 .. 1);
   Bool                    : Boolean;
   Omega                   : String(1 .. 10000)       := (others => ' ');
   K                       : One_String_Type          := (others => ' ');
   Curent_Position_In_Omega : Integer range 1 .. 10000 := 1;
   -- Codes are never longer than
   -- 12 bits; therefore, entry 4095 is the last entry of the LZW table.
   Item : Lzw_Data.Modulo_16_Type range 0 .. 4095;
   use type Lzw_Data.Modulo_16_Type;
   --|
   --|               C O M P R E S S I O N
   --|               ---------------------
   --|
   begin
   --|
   --| initialize input and output files
   --|
   Bool := Lzw_Io_Compress.Init_Reading(File_Name          => The_File);
   Bool := Bool and Lzw_Io_Compress.Init_Writing(File_Name => Into_The_File);
   --|
   --| initialize hash and related variables
   --|
   Initialize;
   --|
   --| omega <- fist char in file to be compressed
   --|
   Omega(1) := Lzw_Io_Compress.Value;
   Lzw_Io_Compress.Next_Character;
   --|
   --| while not end of file loop
   --|
   while not Lzw_Io_Compress.Reading_Over loop
      --|
      --| k <- next char in file
      --|
      K(1) := Lzw_Io_Compress.Value;
      --|
      --| if  exists( omega + k)
      --| omega <- omega + k
      --|
      if Exists(Omega(1 .. Curent_Position_In_Omega) & K)
         then
         Curent_Position_In_Omega        := 1 + Curent_Position_In_Omega;
         Omega(Curent_Position_In_Omega) := K(1);
         --|
         --| else not found read, write new  and reset
         --|
         else
         --|
         --| output read(omega)
         --|
         Item := Read(Omega(1 .. Curent_Position_In_Omega));

         Lzw_Io_Compress.Write(Item           => Item,
                               Number_Of_Bits => Current_Number_Of_Bits);
         --|
         --| add ( omega + k)
         --|
         Add(Omega(1 .. Curent_Position_In_Omega) & K);
         --|
         --| omega <- k
         --|
         Omega(1 .. 1)            := K;
         Curent_Position_In_Omega := 1;
         --|
         --| end if
         --|
      end if;
      --|
      --|   get next character
      --|
      Lzw_Io_Compress.Next_Character;
   end loop;
   --|
   --| output read(omega)
   --|
   Item := Read(Omega(1 .. Curent_Position_In_Omega));

   Lzw_Io_Compress.Write(Item           => Item,
                         Number_Of_Bits => Current_Number_Of_Bits);
   --|
   --|  close
   --|
```

```
      Lzw_Io_Compress.Flush_Buffer_And_Close_File;

    end Compress;

  end Lzw_1;
```

16.3.3 Fichier Lzw_2.ads

```
  with Lzw_Data;
  with Ada.Unchecked_Deallocation;
  --|
  --|
  --|                    ____________________________________
  --|                   ||                                  ||
  --|                   ||                                  ||
  --|                   ||              Lzw_2               ||
  --|                   ||                                  ||
  --|                   ||__________________________________||
  --|
  --|
  package Lzw_2
    is
    --|
    --|                         |==================================|
    --|                         |              Expand              |
    --|                         |==================================|
    --|
    procedure Expand(The_File      : in     String;
                     Into_The_File : in     String);


    private
    use type Lzw_Data.Modulo_16_Type;
    Current_Index          : Lzw_Data.Modulo_16_Type      := 0;
    Current_Number_Of_Bits : Lzw_Data.Number_Of_Bits_Type := Lzw_Data.Number_Of_Bits_Type'First;
    --|
    --|  data as a pointer to a string
    --|
    type String_Access_Type is access all String;
    ---Codes are never longer than
    -- 12 bits; therefore, entry 4095 is the last entry of the LZW table.
    type Table_Type is array (Lzw_Data.Modulo_16_Type range 0 .. 2 ** 12 - 1) of String_Access_Type;
    Table : Table_Type := (others => null);
    --|
    --|  used in delete
    --|
    procedure Free is new Ada.Unchecked_Deallocation(Object => String,
                                                    Name   => String_Access_Type);
  end Lzw_2;
```

16.3.4 Fichier Lzw_2.adb

```
with Ada.Text_Io;
  with Lzw_Io_Expand;
  --|
  --|
  --|                    ____________________________________
  --|                   ||                                  ||
  --|                   ||                                  ||
  --|                   ||              Lzw_2               ||
  --|                   ||                                  ||
  --|                   ||__________________________________||
  --|
  --|
  package body Lzw_2
    is
    --|
    --|                         |==================================|
    --|                         |     Append_To_Table              |
    --|                         |==================================|
    --|
    procedure Append_To_Table(X : in     String)
       --| append_to_table( string) updating last used index
       is
       begin
       --|
       --|  load and increment index
       --|
       Table(Current_Index) := new String'(X);
       Current_Index        := 1 + Current_Index;

       -- The first
       -- output code that is 10 bits long is the one following the creation of table entry
       -- 511, and similarly for 11 (1023) and 12 (2047) bits.
       --|
       --|    set Current_Number_Of_Bits accordingly
       --|
       if Current_Index = 511 or else Current_Index = 1023 or else Current_Index = 2047
          then
          Current_Number_Of_Bits := 1 + Current_Number_Of_Bits;
       end if;
    end Append_To_Table;
```

```
--|
--|                          |==================================|
--|                          |          Initialize              |
--|                          |==================================|
--|
procedure Initialize
   is
   --| initialize:
   --| append ascii code 0 to 255 to a 'table' of strings with indexes 0 to 255
   --| keep last used index
   subtype One_String_Type is String(1 .. 1);
   One_String : One_String_Type;
   begin
   --|
   --|  init table data structures
   --|
   for I in Table'Range loop
      Free(Table(I));
   end loop;
   --|
   --| add as strings the characters from ascii code 0 to ascii code 255
   --|
   for Ascii_Code in 0 .. 255 loop
      One_String(1) := Character'Val(Ascii_Code);
      Append_To_Table(X => One_String);
   end loop;
   --|
   --| 256 and 257 reserved  for clear table and eod: skip them
   --|
   Current_Index          := 258;
   Current_Number_Of_Bits := 9;

end Initialize;
--|
--|                          |==================================|
--|                          |           Read_Table             |
--|                          |==================================|
--|
function Read_Table(Index : in     Lzw_Data.Modulo_16_Type)
   return String
   is
   --| read_table(index) return string
   begin
   --|
   --| return data
   --|
   return Table(Index).all;
end Read_Table;
--|
--|                          |==================================|
--|                          |              Exists              |
--|                          |==================================|
--|
function Exists(Index : in     Lzw_Data.Modulo_16_Type)
   return Boolean
   is
   --| exists(index) return boolean
   begin
   --|
   --| check if not empty
   --|
   return Table(Index) /= null;
end Exists;
--|
--|                          |==================================|
--|                          |              Expand              |
--|                          |==================================|
--|
procedure Expand(The_File      : in     String;
                 Into_The_File : in     String)
   is
   --|
   --|                          |==================================|
   --|                          |              Output              |
   --|                          |==================================|
   --|
   procedure Output(S : in     String)
      is
      begin
      --|
      --| put char on file one by one
      --|
      for I in S'Range loop
         Lzw_Io_Expand.Write(Item => S(I));
      end loop;
   end Output;

   Bool : Boolean;
   ---Codes are never longer than
   -- 12 bits; therefore, entry 4095 is the last entry of the LZW table.
   Incode  : Lzw_Data.Modulo_16_Type range 0 .. 4095;
   Oldcode : Lzw_Data.Modulo_16_Type range 0 .. 4095;
   use type Lzw_Data.Modulo_16_Type;
   --|
   --|                      D E C O M P R E S S I O N
   --|                      -------------------------
   begin
```

```
      --|
      --| initialize the first time is with 9 bits
      --|

      Bool := Lzw_Io_Expand.Init_Reading(File_Name      => The_File,
                                         Number_Of_Bits => 9);

      Bool := Bool and Lzw_Io_Expand.Init_Writing(File_Name => Into_The_File);
      --|
      --|  init table and associated variables
      --|
      Initialize;
      --|
      --|  clear clear_code in front
      --|
      Lzw_Io_Expand.Next_Item(Number_Of_Bits => 9);
      --|
      --| incode <- first 9 bits in file
      --|
      Incode := Lzw_Io_Expand.Value;
      Lzw_Io_Expand.Next_Item(Number_Of_Bits => 9);
      --|
      --| oldcode <- incode
      --|
      Oldcode := Incode;
      --|
      --| output <- read_table(incode)
      --|
      Output(S => Read_Table(Incode));
      --|
      --| while 9 -12 bits integers to read from file loop
      --|
      while not Lzw_Io_Expand.Reading_Over loop
         --|
         --| incode <- input from file (next 9 bits)
         --|
         Incode := Lzw_Io_Expand.Value;
         --|
         --| over exit
         --|
         exit when Incode = Lzw_Data.Eod_Marker;
         --|
         --| clear table marker  ? if so init again
         --|
         if Incode = Lzw_Data.Clear_Table_Marker
            then
            Current_Index          := 258;
            Current_Number_Of_Bits := 9;
            --|
            --|  init table data structures
            --|
            for I in 258 .. Table'Last loop
               Free(Table(I));
            end loop;
            Lzw_Io_Expand.Next_Item(Number_Of_Bits => Current_Number_Of_Bits);
            Incode  := Lzw_Io_Expand.Value;
            Oldcode := Incode;
         end if;
         --|
         --| if exists(incode)
         --|
         if Exists(Incode)
            then
            --|
            --| append_to_table ( read_table(oldcode) + first char of read_table(incode) )
            --|
            Append_To_Table(Read_Table(Oldcode)
                            &
                            Read_Table(Incode) (Read_Table(Incode)'First));
            --|
            --| else
            --|
            else
            --|
            --| append_to_table ( read_table(oldcode) + first char of read_table(oldcode) )
            --|
            Append_To_Table(Read_Table(Oldcode)
                            &
                            Read_Table(Oldcode) (Read_Table(Oldcode)'First));
            --|
            --| end if
            --|
         end if;
         --|
         --| output <- read_table(incode)
         --|
         Output(S => Read_Table(Incode));
         --|
         --| oldcode <- incode
         --|
         Oldcode := Incode;
         --|
         --| end loop  get next if any
         --|
         Lzw_Io_Expand.Next_Item(Number_Of_Bits => Current_Number_Of_Bits);
      end loop;
      --|
```

```
      --|
      --|
      Lzw_Io_Expand.Close_Write;
   end Expand;
 end Lzw_2;
```

16.3.5 Fichier Lzw_Data.ads

```
package Lzw_Data
   is
   --|
   --| data types from standard
   --|
   subtype Block_Type is String(1 .. 128);

   type Modulo_8_Type is mod 2 ** 8;
   for Modulo_8_Type'Size use 8;

   type Modulo_16_Type is mod 2 ** 16;
   for Modulo_16_Type'Size use 16;

   type Modulo_32_Type is mod 2 ** 32;
   for Modulo_32_Type'Size use 32;

   subtype Number_Of_Bits_Type is Integer range 9 .. 12;

   Debug_On           : constant Boolean        := False;
   Clear_Table_Marker : constant Modulo_16_Type := 256;
   Eod_Marker         : constant Modulo_16_Type := 257;

 end Lzw_Data;
```

16.3.6 Fichier Lzw_Debug.ads

```
with Lzw_Data;

 package Lzw_Debug
   is

   subtype Base_Type is Integer range 2 .. 16;

   generic
   type Index_Type is mod <>;
   --|
   --|                          |==================================|
   --|                          |        Print_Modulo_Data         |
   --|                          |==================================|
   --|
   procedure Print_Modulo_Data(X    : in     Index_Type;
                               Base : in     Base_Type);

 end Lzw_Debug;
```

16.3.7 Fichier Lzw_Debug.adb

```
with Ada.Text_Io;

 package body Lzw_Debug
   is
   --|
   --|                          |==================================|
   --|                          |        Print_Modulo_Data         |
   --|                          |==================================|
   --|
   procedure Print_Modulo_Data(X    : in     Index_Type;
                               Base : in     Base_Type)
      is
      Mod_Length : constant Integer := X'Size;-- number of   digits in x
      S          : String(1 .. Mod_Length + 4);
      First      : Integer;
      package Mod_32_Io is new Ada.Text_Io.Modular_Io(Index_Type);
      begin
      --|
      --| translate into string   base#xxxxxxxx#
      --|
      Mod_32_Io.Put(Item => X,
                    Base => Base,
                    To   => S);
      --|
      --| get position of last #
      --|
      for I in reverse 1 .. Mod_Length + 3 loop
         if S(I) = '#'
            then
            First := I;
            exit;
         end if;
      end loop;
      --|
```

```
      --| put leading zeros
      --|
      for I in reverse 4 .. First loop
         S(I) := '0';
      end loop;
      --|
      --| to screen
      --|
      Ada.Text_Io.Put(S(4 .. Mod_Length + 3)
                      & ' ');
   end Print_Modulo_Data;
end Lzw_Debug;
```

16.3.8 Fichier Lzw_Io_Compress.ads

```
with Ada.Streams.Stream_Io;
with Lzw_Data;
--|
--|
--|                    ____________________________________
--|                   ||                                  ||
--|                   ||                                  ||
--|                   ||         Lzw_Io_Compress          ||
--|                   ||                                  ||
--|                   ||__________________________________||
--|
--|
package Lzw_Io_Compress
   is
   --|
   --|                          |==================================|
   --|                          |           Init                   |
   --|                          |==================================|
   --|
   function Init_Reading(File_Name : in     String) return Boolean;
   --|
   --|                          |==================================|
   --|                          |           Value                  |
   --|                          |==================================|
   --|
   function Value return Character;
   --|
   --|                          |==================================|
   --|                          |          Over                    |
   --|                          |==================================|
   --|
   function Reading_Over return Boolean;
   --|
   --|                          |==================================|
   --|                          |          Next                    |
   --|                          |==================================|
   --|
   procedure Next_Character;
   --|
   --|                          |==================================|
   --|                          |          Init_Writing            |
   --|                          |==================================|
   --|
   function Init_Writing(File_Name : in     String) return Boolean;
   --|
   --|                          |==================================|
   --|                          |          Write                   |
   --|                          |==================================|
   --|
   procedure Write(Item           : in     Lzw_Data.Modulo_16_Type;
                   Number_Of_Bits : in     Lzw_Data.Number_Of_Bits_Type);
   --|
   --|                          |==================================|
   --|                          |     Flush_Buffer_And_Close_File  |
   --|                          |==================================|
   --|
   procedure Flush_Buffer_And_Close_File;
   --|
   --|
   --|
   File_Not_Found : exception;
   --|
   --|
   --|
   private

   use type Ada.Streams.Stream_Element_Offset;
   use type Ada.Streams.Stream_Io.Count;
   use type Lzw_Data.Modulo_16_Type;


   subtype Position_Type is Integer range 0 .. 31;
   subtype Read_Stream_Data_Type is
   Ada.Streams.Stream_Element_Array(1 .. Ada.Streams.Stream_Element_Offset(128));


   The_File_In     : Ada.Streams.Stream_Io.File_Type;
   The_File_Out    : Ada.Streams.Stream_Io.File_Type;
   File_Stream_Out : Ada.Streams.Stream_Io.Stream_Access;
   Read_Data       : Read_Stream_Data_Type;
```

```
  Modulo_32                 : Lzw_Data.Modulo_32_Type      := 0;
  Free_Position             : Position_Type                := 31;-- position form 0 to 63
  Current_Position_In_Block : Integer                      := 1;
  Current_Character         : Character                    := Ascii.Nul;
  Read_Count                : Ada.Streams.Stream_Io.Count := 1;
  Done                      : Boolean                      := False;
  Last_Block_In_File_Read   : Boolean                      := False;
  Read_Block                : Lzw_Data.Block_Type          := (others => Ascii.Nul);
  Read_Length               : Ada.Streams.Stream_Element_Offset
  := Ada.Streams.Stream_Element_Offset(128);

end Lzw_Io_Compress;
```

16.3.9 Fichier Lzw_Io_Compress.adb

```
with Ada.Text_Io;
with Lzw_Data;
with Unchecked_Conversion;
with Lzw_Debug;
--|
--|
--|                    _________________________________
--|                   ||                                ||
--|                   ||                                ||
--|                   ||      Lzw_Io_Compress           ||
--|                   ||                                ||
--|                   ||________________________________||
--|
--|
package body Lzw_Io_Compress
  is
  --|
  --|  instantiations for debug
  --|
  procedure Print_Bits is new Lzw_Debug.Print_Modulo_Data(Index_Type => Lzw_Data.Modulo_8_Type);
  procedure Print_Bits is new Lzw_Debug.Print_Modulo_Data(Index_Type => Lzw_Data.Modulo_16_Type);
  procedure Print_Bits is new Lzw_Debug.Print_Modulo_Data(Index_Type => Lzw_Data.Modulo_32_Type);

  --***************************************************************
  --******************     R E A D I N G    *************************
  --***************************************************************
  --|
  --|                          |==================================|
  --|                          |       Get_Next_Block             |
  --|                          |==================================|
  --|
  procedure Get_Next_Block
     is
     --|
     --| this function assigns the same memory adress to the Block_Type type substring
     --| and Word_Array_type and return the corresponding Word_Array_type value
     --|
     function Data_To_Block is new Unchecked_Conversion(Source => Read_Stream_Data_Type,
                                                        Target => Lzw_Data.Block_Type);

     begin
     --|
     --| read next if any ( if none length =0 if partly empty length <64 if Filled length=64
     --|
     Ada.Streams.Stream_Io.Read(File => The_File_In,
                                Item => Read_Data,
                                Last => Read_Length,
                                From => Read_Count);

     Read_Block := Data_To_Block(Read_Data);
     Read_Count := Ada.Streams.Stream_Io.Count(128) + Read_Count;
     --|
     --| if over update and set done to true
     --|
     if Integer(Read_Length) < 128 -- done
        then
        Last_Block_In_File_Read := True;
     end if;

     --=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+
     --=-+=-+=-+=-+=-+=-+=   S T A R T   D E B U G   =-+=-+=-+=-+=-+=-+
     --=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+
     if Lzw_Data.Debug_On
        then
        Ada.Text_Io.New_Line;
        Ada.Text_Io.Put_Line("Lzw_Io.Get_Next_Block"
                             & Integer'Image(Integer(Read_Count))
                             & Integer'Image(Integer(Read_Length)));
     end if;
     --=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+
     --=-+=-+=-+=-+=-+=-+=    E N D    D E B U G     =-+=-+=-+=-+=-+=-+
     --=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+
  end Get_Next_Block;
  --|
  --|                          |==================================|
  --|                          |         Next                     |
  --|                          |==================================|
  --|
  procedure Next_Character
```

```
   is
   begin
   Done := Last_Block_In_File_Read
   and Current_Position_In_Block = 1 + Integer(Read_Length);

   if not Done
      then
      Current_Character         := Read_Block(Current_Position_In_Block);
      Current_Position_In_Block := 1 + Current_Position_In_Block;

      if Current_Position_In_Block > 128
         then
         Current_Position_In_Block := 1;
         Get_Next_Block;
      end if;

   end if;
end Next_Character;
--|
--|                         |==================================|
--|                         |           Init                   |
--|                         |==================================|
--|
function Init_Reading(File_Name : in     String)
   return Boolean
   is
   begin
   --|
   --| close if previous file already used
   --|
   if Ada.Streams.Stream_Io.Is_Open(File => The_File_In)
      then
      Ada.Streams.Stream_Io.Close(File => The_File_In);
   end if;
   --|
   --| set to ascii.nul, zero and false
   --|
   Current_Character := Ascii.Nul;
   Done              := False;
   Read_Count        := 1;
   --|
   --| try opening the file
   --|
   Ada.Streams.Stream_Io.Open(File => The_File_In,
                              Name => File_Name,
                              Mode => Ada.Streams.Stream_Io.In_File);

   --|
   --| success get first block and init first char if any :
   --|
   Get_Next_Block;
   Next_Character;
   --|
   --| file opened
   --|
   return True;
   --|
   --|   no such file ??
   --|
   exception
   when others =>
   Done := True;
   Ada.Streams.Stream_Io.Close(File => The_File_In);

   return False;
end Init_Reading;
--|
--|                         |==================================|
--|                         |           Value                  |
--|                         |==================================|
--|
function Value
   return Character
   is
   begin
   return Current_Character;
end Value;
--|
--|                         |==================================|
--|                         |         Over                     |
--|                         |==================================|
--|
function Reading_Over
   return Boolean
   is
   begin
   return Done;
end Reading_Over;

--*****************************************************************
--*****************  W R I T I N G ********************************
--*****************************************************************

--|
--|                         |==================================|
--|                         |           Init                   |
--|                         |==================================|
```

```
--|
function Init_Writing(File_Name : in     String)
   return Boolean
   is
   begin
   --|
   --| close if previous file already used
   --|
   if Ada.Streams.Stream_Io.Is_Open(File => The_File_Out)
      then
      Ada.Streams.Stream_Io.Close(File => The_File_Out);
   end if;
   --|
   --| try opening the file
   --|
   Ada.Streams.Stream_Io.Create(File => The_File_Out,
                                Name => File_Name,
                                Mode => Ada.Streams.Stream_Io.Out_File);
   --|
   --| get stream access
   --|
   File_Stream_Out := Ada.Streams.Stream_Io.Stream(File => The_File_Out);
   --|
   --| file created
   --|
   return True;
   --|
   --|   creation problem ??
   --|
   exception
   when others =>
   Done := True;
   return False;
end Init_Writing;
--|
--|                          |==================================|
--|                          |         Write                    |
--|                          |==================================|
--|
procedure Write(Item          : Lzw_Data.Modulo_16_Type;
                Number_Of_Bits : Lzw_Data.Number_Of_Bits_Type)
   is
   --|
   --|
   --| for example the following six   9 bit data followed by a flush_buffer
   --| 111111111
   --| 000000000
   --| 111111111
   --| 000000000
   --| 111111111
   --| 000000000
   --| flush_buffer
   --|
   --| will be stored as
   --| 111111111000000000111111111100000000011111111100000000
   --| and as bytes
   --| 11111111 10000000 00111111 11100000 00001111 1111100 00000000 .........
   --|                                                                 ^
   --|             From the flush buffer command                       |
   --|
   --|
   --|
   --|   needed,for arithmetic
   --|
   use type Lzw_Data.Modulo_8_Type;
   use type Lzw_Data.Modulo_16_Type;
   use type Lzw_Data.Modulo_32_Type;
   --|
   --| buffer and word to write
   --|
   Byte_To_Write_On_File : Lzw_Data.Modulo_8_Type;
   Temporary_Storage     : Lzw_Data.Modulo_32_Type;

   begin
   --|
   --| put item into temp
   --|
   Temporary_Storage := Lzw_Data.Modulo_32_Type(Item);
   --|
   --| move bits left
   --|
   Temporary_Storage := Temporary_Storage * 2 ** (Free_Position - Number_Of_Bits + 1);
   --|
   --| "ADD" these bits
   --|
   Modulo_32 := Modulo_32 or Temporary_Storage;
   --|
   --|  update free pos for next time
   --|
   Free_Position := Free_Position - Number_Of_Bits;
   --|
   --| enought bits to write ( at least 8) ???    the loop is there
   --| because sometimes there are 16 bits available
   --|
   while Free_Position < 23 loop
      --|
      --|shift right to keep only 8 bits
```

```
         --|
         Temporary_Storage := Modulo_32 / 2 ** 24;
         --|
         --| load
         --|
         Byte_To_Write_On_File := Lzw_Data.Modulo_8_Type(Temporary_Storage);
         --|
         --| write
         --|
         Lzw_Data.Modulo_8_Type'Write(File_Stream_Out,
                                      Byte_To_Write_On_File);
         --|
         --| update
         --|
         Free_Position := Free_Position + 8;
         --|
         --| shift left to clear already written bits
         --|
         Modulo_32 := Modulo_32 * 2 ** 8;

         --=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+
         --=-+=-+=-+=-+=-+=-+=   S T A R T   D E B U G    =-+=-+=-+=-+=-+=-+
         --=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+
         if Lzw_Data.Debug_On
            then
            Ada.Text_Io.Put(" ¯ Lzw_Io_Compress.Write  Byte_to_write_on_file ");
            Print_Bits(X    => Byte_To_Write_On_File,
                       Base => 2);
            Ada.Text_Io.Put(" ± Free_Position"
                            & Integer'Image(Free_Position));
            Ada.Text_Io.New_Line;
         end if;
         --=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+
         --=-+=-+=-+=-+=-+=-+=    E N D    D E B U G      =-+=-+=-+=-+=-+=-+
         --=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+
      end loop;

      --Print_Bits(X => Modulo_32,base=>2);

   end Write;
   --|
   --|                        |===================================|
   --|                        |     Flush_Buffer_And_Close_File   |
   --|                        |===================================|
   --|
   procedure Flush_Buffer_And_Close_File
      is
      begin
      --|
      --|  put eod and therefore force write of last stored (but not yet written) bits
      --|
      Write(Item           => Lzw_Data.Eod_Marker,
            Number_Of_Bits => 9);
      --|
      --| close files
      --|
      Ada.Streams.Stream_Io.Close(File => The_File_Out);
      Ada.Streams.Stream_Io.Close(File => The_File_In);

   end Flush_Buffer_And_Close_File;

 end Lzw_Io_Compress;
```

16.3.10 Fichier Lzw_Io_Expand.ads

```
 with Ada.Streams.Stream_Io;
 with Lzw_Data;
 --|
 --|
 --|                 _________________________________
 --|                ||                               ||
 --|                ||                               ||
 --|                ||          Lzw_Io_Expand        ||
 --|                ||                               ||
 --|                ||_______________________________||
 --|
 --|
 package Lzw_Io_Expand
   is
   --|
   --|                        |===================================|
   --|                        |              Init                 |
   --|                        |===================================|
   --|
   function Init_Reading(File_Name      : in     String;
                         Number_Of_Bits : in     Lzw_Data.Number_Of_Bits_Type) return Boolean;
   --|
   --|                        |===================================|
   --|                        |              Value                |
   --|                        |===================================|
   --|
   function Value return Lzw_Data.Modulo_16_Type;
   --|
   --|                        |===================================|
   --|                        |            Over                   |
```

```
   --|                        |================================|
   --|
   function Reading_Over return Boolean;
   --|
   --|                        |================================|
   --|                        |         Next                   |
   --|                        |================================|
   --|
   procedure Next_Item(Number_Of_Bits : in     Lzw_Data.Number_Of_Bits_Type);
   --|
   --|                        |================================|
   --|                        |         Init_Writing           |
   --|                        |================================|
   --|
   function Init_Writing(File_Name : in     String) return Boolean;
   --|
   --|                        |================================|
   --|                        |         Write                  |
   --|                        |================================|
   --|
   procedure Write(Item : in     Character);
   --|
   --|                        |================================|
   --|                        |      Close_Write               |
   --|                        |================================|
   --|
   procedure Close_Write;
   --|
   --|
   --|
   File_Not_Found : exception;
   --|
   --|
   --|
   private
   subtype Position_Type is Integer range 0 .. 31;

   The_File_In     : Ada.Streams.Stream_Io.File_Type;
   The_File_Out    : Ada.Streams.Stream_Io.File_Type;
   File_Stream_In  : Ada.Streams.Stream_Io.Stream_Access;
   File_Stream_Out : Ada.Streams.Stream_Io.Stream_Access;

   Done          : Boolean                 := False;
   Item          : Lzw_Data.Modulo_16_Type := 0;
   Modulo_32     : Lzw_Data.Modulo_32_Type := 0;
   Free_Position : Position_Type           := 31;-- position from 0 to 31

 end Lzw_Io_Expand;
```

16.3.11 Fichier Lzw_Io_Expand.adb

```
with Ada.Text_Io;
  with Ada.Streams.Stream_Io;
  with Ada.Io_Exceptions;
  with Lzw_Data;
  with Lzw_Debug;
  --|
  --|
  --|                    ____________________________________
  --|                   ||___________________________________||
  --|                   ||                                   ||
  --|                   ||         Lzw_Io_Expand             ||
  --|                   ||                                   ||
  --|                   ||___________________________________||
  --|                    ____________________________________
  --|
  --|
  package body Lzw_Io_Expand
    is
    --|
    --|   needed,for arithmetic
    --|
    use type Lzw_Data.Modulo_8_Type;
    use type Lzw_Data.Modulo_16_Type;
    use type Lzw_Data.Modulo_32_Type;
    --|
    --|  instantiations for debug
    --|
    procedure Print_Bits is new Lzw_Debug.Print_Modulo_Data(Index_Type => Lzw_Data.Modulo_8_Type);
    procedure Print_Bits is new Lzw_Debug.Print_Modulo_Data(Index_Type => Lzw_Data.Modulo_16_Type);
    procedure Print_Bits is new Lzw_Debug.Print_Modulo_Data(Index_Type => Lzw_Data.Modulo_32_Type);

    --*****************************************************************
    --******************     R E A D I N G   ***********************
    --*****************************************************************

    --|
    --|                        |================================|
    --|                        |           Value                |
    --|                        |================================|
    --|
    function Value
       return Lzw_Data.Modulo_16_Type
       is
       begin
       return Item;
```

```
end Value;
--|
--|                          |==================================|
--|                          |          Over                    |
--|                          |==================================|
--|
function Reading_Over
   return Boolean
   is
   begin
   return Done;
end Reading_Over;
--|
--|                          |==================================|
--|                          |        Next_Item                 |
--|                          |==================================|
--|
procedure Next_Item(Number_Of_Bits : in     Lzw_Data.Number_Of_Bits_Type)
   is

   Byte_Read_From_File : Lzw_Data.Modulo_8_Type;
   Temporary_Storage   : Lzw_Data.Modulo_32_Type;
   begin
   --|
   --| read enough bits to have a least Number_Of_Bits available
   --|
   while Free_Position > 31 - Number_Of_Bits loop
      --|
      --| read
      --|
      Lzw_Data.Modulo_8_Type'Read(File_Stream_In,
                                  Byte_Read_From_File);
      --|
      --| put item into temp
      --|
      Temporary_Storage := Lzw_Data.Modulo_32_Type(Byte_Read_From_File);
      --|
      --| move 8 bits left
      --|
      Temporary_Storage := Temporary_Storage * 2 ** (Free_Position - 7);
      --|
      --| "ADD" these bits
      --|
      Modulo_32 := Modulo_32 or Temporary_Storage;
      --|
      --|  update free pos for next time
      --|
      Free_Position := Free_Position - 8;
   end loop;
   --|
   --|shift right to keep only Number_Of_Bits bits
   --|
   Temporary_Storage := Modulo_32 / 2 ** (32 - Number_Of_Bits);
   --|
   --| load
   --|
   Item := Lzw_Data.Modulo_16_Type(Temporary_Storage);
   --|
   --| update
   --|
   Free_Position := Free_Position + Number_Of_Bits;
   --|
   --|  shift left to clear already written bits
   --|
   Modulo_32 := Modulo_32 * 2 ** Number_Of_Bits;

   --=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+
   --=-+=-+=-+=-+=-+=-+=  S T A R T   D E B U G   =-+=-+=-+=-+=-+=-+=-+
   --=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+
   if Lzw_Data.Debug_On
      then
      Ada.Text_Io.Put("Item ");
      Print_Bits(X    => Item,
                 Base => 2);

      Ada.Text_Io.Put("222  Free_Position"
                      & Integer'Image(Free_Position));
      Ada.Text_Io.New_Line;
   end if;
   --=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+
   --=-+=-+=-+=-+=-+=-+=   E N D    D E B U G     =-+=-+=-+=-+=-+=-+=-+
   --=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+=-+
   exception
   when Ada.Io_Exceptions.End_Error =>
   Done := True;
   Ada.Streams.Stream_Io.Close(File => The_File_In);
end Next_Item;
--|
--|                          |==================================|
--|                          |           Init                   |
--|                          |==================================|
--|
function Init_Reading(File_Name      : in     String;
                      Number_Of_Bits : in     Lzw_Data.Number_Of_Bits_Type)
   return Boolean
   is
   begin
```

```
      --|
      --| close if previous file already used
      --|
      if Ada.Streams.Stream_Io.Is_Open(File => The_File_In)
         then
         Ada.Streams.Stream_Io.Close(File => The_File_In);
      end if;
      --|
      --| try opening the file
      --|
      Ada.Streams.Stream_Io.Open(File => The_File_In,
                                 Name => File_Name,
                                 Mode => Ada.Streams.Stream_Io.In_File);
      --|
      --| get stream access
      --|
      File_Stream_In := Ada.Streams.Stream_Io.Stream(File => The_File_In);
      --|
      --| load first data
      --|
      Next_Item(Number_Of_Bits => Number_Of_Bits);
      --|
      --| No problem
      --|
      return True;
      --|
      --|   creation problem ??
      --|
      exception
      when others =>
      Done := True;
      return False;

   end Init_Reading;


   --*****************************************************************
   --******************  W R I T I N G ***********************************
   --*****************************************************************



   --|
   --|                          |=================================|
   --|                          |         Init_Writing            |
   --|                          |=================================|
   --|
   function Init_Writing(File_Name : in     String)
      return Boolean
      is
      begin
      --|
      --| close if previous file already used
      --|
      if Ada.Streams.Stream_Io.Is_Open(File => The_File_Out)
         then
         Ada.Streams.Stream_Io.Close(File => The_File_Out);
      end if;
      --|
      --| try opening the file
      --|
      Ada.Streams.Stream_Io.Create(File => The_File_Out,
                                   Name => File_Name,
                                   Mode => Ada.Streams.Stream_Io.Out_File);
      --|
      --| get stream access
      --|
      File_Stream_Out := Ada.Streams.Stream_Io.Stream(File => The_File_Out);
      --|
      --| file created
      --|
      return True;
      --|
      --|   creation problem ??
      --|
      exception
      when others =>
      Done := True;
      return False;
   end Init_Writing;
   --|
   --|                          |=================================|
   --|                          |         Write                   |
   --|                          |=================================|
   --|
   procedure Write(Item : in     Character)
      is
      begin
      --|
      --| write
      --|
      Lzw_Data.Modulo_8_Type'Write(File_Stream_Out,
                                   Character'Pos(Item));
   end Write;
   --|
   --|                          |=================================|
   --|                          |      Close_Write                |
```

```
   --|                        |================================|
   --|
   procedure Close_Write
      is
      begin
      Ada.Streams.Stream_Io.Close(File => The_File_Out);

   end Close_Write;
end Lzw_Io_Expand;
```

16.3.12 Fichier Lzw.adb

```
--|   UNIVERSITY TEACHING CLASS 2001, 2002, 2003, 2004, 2005 & 2006
--|    "ALGORITHMS USING ADVANCED ADA"
--|
--|Daniel.Gaudry@wanadoo.fr
--|
--|   =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
--|   L Z W   C O M P R E S S I O N     A N D    D E C O M P R E S S I O N
--|   =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
--
--  adapted from T.Welsh IEEE Computer June 1984
--  see lzw.pdf
--
--              C O M P R E S S I O N
--              --------------------
--
-- INITIALISATION
--
-- add as strings the characters from ascii code 0 to ascii code 255
-- to an hash table storing at the same time an associated index from 0 to 255
-- the number of bits to store the index is unsigned 9-12 bits STARTING AT 9
-- the last used index number will be kept for future additions to the
-- hash table.
--
-- the hash table interface will only implement the following operations :
--
-- initialize as described above   +
--                  number of bits<-9
--                  index <- 258
--
-- add ( string)  storing the string and the associated incremented index
-- checking for index to be 511 , 1023 or   2047 => increment number of bits
-- checking for index uto be 4095 => clear hash and reset number of bits and index
--
-- exists( string) return boolean
-- read(from string) returning the index associated to the string
--
--
--
-- the algorithm is as follows:
--
-- initialize
-- omega <- fist char in file to be compressed
--
-- while not end of file loop
-- check for eod exit if true
-- check for clear and reset hash
-- k <- next char in file
--
-- if  exists( omega + k)
-- omega <- omega + k
-- else
-- output read(omega)  (9-12 bits)
-- add ( omega + k)
-- omega <- k
-- end if
-- end loop
-- output read(omega)  (9-12 bits)
--
--
--                    D E C O M P R E S S I O N
--                    -------------------------
-- initialize:
-- Add ascii code 0 to 255 to a 'table' of strings with indexes 0 to 255
-- number of bits<-9
-- index <- 258
-- keep last used index
--
-- the 'table' interface will only implement the following operations :
--
-- append_to_table( string) updating last used index
-- checking for index to be 511 , 1023 or   2047 => increment number of bits
--
-- read_table(index) return string
-- exists(index) return boolean
--
-- the algorithm is as follows :
--
-- initialize
-- incode <- first 9 bits in file
-- oldcode <- incode
-- output <- read_table(incode)
-- while 9-12 bits integers to read from file loop
-- incode <- input from file (next 9-12 bits)
```

```
-- incode = clear table marker
--          initialize reload incode oldcode and reset index and number of bits
-- if exists(incode)
-- append_to_table ( read_table(oldcode) + first char of (read_table(incode)) )
-- else
-- append_to_table ( read_table(oldcode) + first char of (read_table(oldcode)) )
-- end if
-- output <- read_table(incode)
-- oldcode <- incode
-- end loop
--
--                  M O R E   D E T A I L S   see PDFReference16.pdf page 40-60
--
-- Details of LZW Encoding:
--
-- Data encoded using the LZW compression method consists of a sequence of
-- codes that are 9 to 12 bits long. Each code represents a single character of input
-- data (0-255), a clear-table marker (256), an EOD marker (257), or a table entry
-- representing a multiple-character sequence that has been encountered previously
-- in the input (258 or greater).
-- Initially, the code length is 9 bits and the LZW table contains only entries for the
-- 258 fixed codes. As encoding proceeds, entries are appended to the table, associating
-- new codes with longer and longer sequences of input characters. The
-- encoder and the decoder maintain identical copies of this table.
-- Whenever both the encoder and the decoder independently (but synchronously)
-- realize that the current code length is no longer sufficient to represent the number
-- of entries in the table, they increase the number of bits per code by 1. The first
-- output code that is 10 bits long is the one following the creation of table entry
-- 511, and similarly for 11 (1023) and 12 (2047) bits. Codes are never longer than
-- 12 bits; therefore, entry 4095 is the last entry of the LZW table.
-- The encoder executes the following sequence of steps to generate each output
-- code:
-- 1. Accumulate a sequence of one or more input characters matching a sequence
-- already present in the table. For maximum compression, the encoder looks for
-- the longest such sequence.
-- 2. Emit the code corresponding to that sequence.
-- 3. Create a new table entry for the first unused code. Its value is the sequence
-- found in step 1 followed by the next input character.
-- For example, suppose the input consists of the following (hex) sequence of ASCII character
-- codes:
-- 45 45 45 45 45 65 45 45 45 66    corresponding to EEEEEeEEEf
-- Starting with an empty table, the encoder proceeds as shown in Table


-- TABLE               Typical LZW encoding sequence
--
-- INPUT SEQUENCE    OUTPUT CODE          CODE ADDED TO TABLE    SEQUENCE REPRESENTED BY NEW CODE
--                    256 (clear-table)
-- 45                  45                        258                        45 45
-- 45 45               258                       259                        45 45 45
-- 45 45               258                       260                        45 45 65
-- 65                  65                        261                        65 45
-- 45 45 45            259                       262                        45 45 45 66
-- 66                  66
--                    257 (EOD)

-- Codes are packed into a continuous bit stream, high-order bit first. This stream is
-- then divided into 8-bit bytes, high-order bit first. Thus, codes can straddle byte
-- boundaries arbitrarily. After the EOD marker (code value 257), any leftover bits
-- in the final byte are set to 0.
-- In the example above, all the output codes are 9 bits long; they would pack into
-- bytes as follows (represented in hexadecimal):
-- 80 0B 60 50 22 0C 0C 85 01
-- To adapt to changing input sequences, the encoder may at any point issue a cleartable
-- code, which causes both the encoder and the decoder to restart with initial
-- tables and a 9-bit code length. By convention, the encoder begins by issuing a
-- clear-table code. It must issue a clear-table code when the table becomes full; it
-- may do so sooner.

with Lzw_1;
with Lzw_2;
with Ada.Text_Io;
--|
--|               |________________|
--|               |                |
--|               |      Lzw       |
--|               |________________|
--|               |                |
--|
procedure Lzw
  is
  begin


  Ada.Text_Io.Put_Line("compressing x.txt into y.txt ");

  Lzw_1.Compress(The_File      => "x.txt",
                 Into_The_File => "y.txt");

  Ada.Text_Io.Put_Line("expanding   y.txt into z.txt");

  Lzw_2.Expand(The_File      => "y.txt",
               Into_The_File => "z.txt");
end Lzw;
```

16.3.13 Fichier Linked_Hash.ads

```
with Ada.Unchecked_Deallocation;
  --|
  --|
  --|  needed for hash_of availability
  --|
  generic

  type Data_Type is private;


  --|
  --| for = operator
  --|
  with function "=" (Left  : in Data_Type;
                     Right : in     Data_Type)
    return Boolean
  is <>;

  package Linked_Hash
    is
    Record_Not_Found_On_Delete                    : exception;
    type Linked_Hash_Table_Type(Number_Of_Buckets : Positive) is limited private;
    type Iterator_Type is limited private;

    --|
    --|                       |_____________________________|
    --|                       |                             |
    --|                       |              H_Of           |
    --|                       |_____________________________|
    --|                       |                             |
    --|
    function H_Of(The_String : in     String)
    return Integer;
    --|
    --|                       |_____________________________|
    --|                       |                             |
    --|                       |              Seek           |
    --|                       |_____________________________|
    --|                       |                             |
    --|
    procedure Seek(Data              : in     Data_Type;
                   In_The_Hash_Table : in     Linked_Hash_Table_Type;
                   Success           :    out Boolean;
                   Found_Record      : in out Data_Type;
                   Bucket_Number     : in     Natural);
    --|
    --|                       |_____________________________|
    --|                       |                             |
    --|                       |              Add            |
    --|                       |_____________________________|
    --|                       |                             |
    --|
    procedure Add(Data              : in     Data_Type;
                  To_The_Hash_Table : in out Linked_Hash_Table_Type;
                  Bucket_Number     : in     Natural);
    --|
    --|                       |_____________________________|
    --|                       |                             |
    --|                       |              Delete         |
    --|                       |_____________________________|
    --|                       |                             |
    --|
    procedure Delete(Data              : in     Data_Type;
                     In_The_Hash_Table : in out Linked_Hash_Table_Type;
                     Bucket_Number     : in     Natural);
    --|
    --|                       |_____________________________|
    --|                       |                             |
    --|                       |           Size_Of           |
    --|                       |_____________________________|
    --|                       |                             |
    --|
    function Size_Of(The_Hash_Table : in     Linked_Hash_Table_Type)
    return Integer;
    --|
    --|                       |_____________________________|
    --|                       |                             |
    --|                       |            Initialize       |
    --|                       |_____________________________|
    --|                       |                             |
    --|
    procedure Initialize(Iterator           :    out Iterator_Type;
                         For_The_Hash_Table : in out Linked_Hash_Table_Type);
    --|
    --|                       |_____________________________|
    --|                       |                             |
    --|                       |           Is_Done           |
    --|                       |_____________________________|
    --|                       |                             |
    --|
    function Is_Done(Iterator : in     Iterator_Type)
    return Boolean;
    --|
    --|                       |_____________________________|
```

```
   --|                     |                                  |
   --|                     |      Current_Value               |
   --|                     |__________________________________|
   --|                     |                                  |
   --|
   function Current_Value(Iterator : in     Iterator_Type)
   return Data_Type;
   --|
   --|                     |__________________________________|
   --|                     |                                  |
   --|                     |      To_Next_Value               |
   --|                     |__________________________________|
   --|                     |                                  |
   --|
   procedure To_Next_Value(Iterator : in out Iterator_Type);
   --|                     |__________________________________|
   --|                     |                                  |
   --|                     |         Initialize               |
   --|                     |__________________________________|
   --|                     |                                  |
   --|
   procedure Initialize(Hash_Table : in out Linked_Hash_Table_Type);
   --|
   --|                     |__________________________________|
   --|                     |                                  |
   --|                     |       Print_Statistics           |
   --|                     |__________________________________|
   --|                     |                                  |
   --|
   procedure Print_Statistics(Hash_Table : in     Linked_Hash_Table_Type);


   private

   type Node_Type;
   --|
   --|
   --|
   type Node_Access_Type is access all Node_Type;
   --|
   --|
   --|
   type Node_Type is
      record
      Data           : Data_Type;
      Access_To_Next : Node_Access_Type := null;
   end record;
   --|
   --|  used in delete
   --|
   procedure Free_Node is new Ada.Unchecked_Deallocation(Object => Node_Type,
                                                         Name   => Node_Access_Type);
   --|
   --|  hash table
   --|
   type Hash_Table_Array_Type is array (Integer range <>) of Node_Access_Type;
   --|
   --|
   --|
   type Linked_Hash_Table_Type(Number_Of_Buckets : Positive)
      is
      record
      Hash_Table_Array : aliased Hash_Table_Array_Type(0 .. Number_Of_Buckets) := (others => null);
      How_Many_Stored  : Natural                                               := 0;
   end record;
   --|
   --|  needed for the iterator
   --|
   type Hash_Table_Array_Access_Type is access all Hash_Table_Array_Type;
   --|
   --|  iterator
   --|
   type Iterator_Type is
      record
      Current_Node_Access     : Node_Access_Type             := null;
      Hash_Table_Index        : Integer                      := 0;
      Number_Of_Buckets       : Integer                      := 0;
      Hash_Table_Array_Access : Hash_Table_Array_Access_Type := null;
   end record;

   type Statistics_Type is
      record
      Add_Calls    : Integer := 0;
      Delete_Calls : Integer := 0;
      Seek_Calls   : Integer := 0;

   end record;
   Statistics : Statistics_Type;
 end Linked_Hash;
```

16.3.14 Fichier Linked_Hash.adb

```
with Ada.Text_Io;
```

```
package body Linked_Hash
  is
  --|
  --|                     |_____________________________|
  --|                     |                             |
  --|                     |            H_Of             |
  --|                     |_____________________________|
  --|                     |                             |
  --|
  function H_Of(The_String : in     String)
     return Integer
     is
     --|
     --| 32 bits avec fonctions logiques   or and xor ,.....
     --|
     type H_Type is mod 2 ** 32;
     --|
     --| H <- 0   z <- 0
     --|
     H : H_Type := H_Type'First;
     Z : H_Type := H_Type'First;

     procedure Debug_Hexadecimal(X : in     H_Type)
        is
        package Iio is new Ada.Text_Io.Modular_Io(H_Type);
        begin
        Iio.Put(Item  => X,
                Base  => 16,
                Width => 12);
        Ada.Text_Io.Put(' ');
     end Debug_Hexadecimal;

     begin
     --Ada.Text_Io.Put('!'&The_String&'!');
     --|
     --| Boucle sur tous les caractï¿œes
     --|
     for I in The_String'Range loop
        --|
        --| Dï¿œaler H de 4 bits vers la gauche
        --|
        H := H * 2 ** 4;
        --|
        --|H <- H + valeur ASCII du caractï¿œe courant
        --|
        H := H + Character'Pos(The_String(I));
        --|
        --|Si l'un des 4 bits de poids le plus ï¿œevï¿œest ï¿œal ï¿œ1
        --|
        if (H and 16#F0000000#) /= 0
           --|
           --|Alors
           --|
           then
           --|
           --|Z <- H
           --|
           Z := H;
           --Ada.Text_Io.Put("Z ");
           --Debug_Hexadecimal(X => Z);
           --|
           --|Dï¿œaler h de 24 bit ( 6 positions hexadecimal) vers la droite
           --|
           H := H / 2 ** 24;
           --Ada.Text_Io.Put("h24 ");
           --Debug_Hexadecimal(X => H);
           --|
           --|H <- H   xor   Z
           --|
           H := H xor Z;
           --Ada.Text_Io.Put("h xor ");
           --Debug_Hexadecimal(X => H);
           --|
           --|Mettre ï¿œ0 les 4 bits de poids le plus ï¿œevï¿œde H
           --|
           H := H and 16#0FFFFFFF#;
           --Ada.Text_Io.Put(" and  ");
           --Debug_Hexadecimal(X => H);
           --Ada.Text_Io.New_Line;
           --|
           --|Fin si
           --|
        end if;
     end loop;
     --|
     --|Renvoi H
     return Integer(H);
  end H_Of;
  --|
  --|                     |_____________________________|
  --|                     |                             |
  --|                     |            Seek             |
  --|                     |_____________________________|
  --|                     |                             |
  --|
  procedure Seek(Data              : in     Data_Type;
                 In_The_Hash_Table : in     Linked_Hash_Table_Type;
```

```
                Success           :    out Boolean;
                Found_Record      : in out Data_Type;
                Bucket_Number     : in     Natural)
   is
   Local_Bucket_Number : Natural          := Bucket_Number mod (1 + In_The_Hash_Table.Number_Of_Buckets);
   Node_Access         : Node_Access_Type := In_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number);
   begin
   Statistics.Seek_Calls := 1 + Statistics.Seek_Calls;
   Success               := False;
   while null /= Node_Access loop
      if Data = Node_Access.all.Data
         then
         Success      := True;
         Found_Record := Node_Access.all.Data;
         return;
      end if;
      Node_Access := Node_Access.Access_To_Next;
   end loop;
end Seek;
--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |                 Add                 |
--|                     |_____________________________________|
--|                     |                                     |
--|
procedure Add(Data              : in     Data_Type;
              To_The_Hash_Table : in out Linked_Hash_Table_Type;
              Bucket_Number     : in     Natural)
   is
   Local_Bucket_Number : Natural := Bucket_Number mod (1 + To_The_Hash_Table.Number_Of_Buckets);
   begin
   Statistics.Add_Calls := 1 + Statistics.Add_Calls;
   --|
   --| insert
   --|
   To_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number)
   := new Node_Type'(Data           => Data,
                     Access_To_Next => To_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number));

   To_The_Hash_Table.How_Many_Stored := 1 + To_The_Hash_Table.How_Many_Stored;
end Add;
--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |                Delete               |
--|                     |_____________________________________|
--|                     |                                     |
--|
procedure Delete(Data              : in     Data_Type;
                 In_The_Hash_Table : in out Linked_Hash_Table_Type;
                 Bucket_Number     : in     Natural)
   is
   Local_Bucket_Number  : Natural          := Bucket_Number mod (1 + In_The_Hash_Table.Number_Of_Buckets);
   Current_Node_Access  : Node_Access_Type := In_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number);
   Previous_Node_Access : Node_Access_Type := In_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number);

   begin
   Statistics.Delete_Calls := 1 + Statistics.Delete_Calls;
   --| find the item to remove
   while null /= Current_Node_Access loop
      exit when Data = Current_Node_Access.all.Data;
      Previous_Node_Access := Current_Node_Access;
      Current_Node_Access  := Current_Node_Access.Access_To_Next;
   end loop;

   if null = Current_Node_Access
      then
      raise Record_Not_Found_On_Delete;
   end if;

   --| first one on linked list
   if Current_Node_Access = Previous_Node_Access
      then
      In_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number)
      := In_The_Hash_Table.Hash_Table_Array(Local_Bucket_Number).Access_To_Next;
      else
      Previous_Node_Access.Access_To_Next := Current_Node_Access.Access_To_Next;
   end if;
   Free_Node(Current_Node_Access);
   In_The_Hash_Table.How_Many_Stored := In_The_Hash_Table.How_Many_Stored - 1;
end Delete;
--|
--|                     |_____________________________________|
--|                     |                                     |
--|                     |             Size_Of                 |
--|                     |_____________________________________|
--|                     |                                     |
--|
function Size_Of(The_Hash_Table : in     Linked_Hash_Table_Type)
   return Integer
   is
   begin
   return The_Hash_Table.How_Many_Stored;
end Size_Of;
--|
--|                     |_____________________________________|
```

```
--|                     |                                  |
--|                     |            Is_Done               |
--|                     |__________________________________|
--|                     |                                  |
--|
function Is_Done(Iterator : in     Iterator_Type)
   return Boolean
   is
   begin
   return Iterator.Hash_Table_Index > Iterator.Number_Of_Buckets;
end Is_Done;
--|
--|                     |__________________________________|
--|                     |                                  |
--|                     |      Current_Value               |
--|                     |__________________________________|
--|                     |                                  |
--|
function Current_Value(Iterator : in     Iterator_Type)
   return Data_Type
   is
   begin
   return Iterator.Current_Node_Access.Data;
end Current_Value;
--|
--|                     |__________________________________|
--|                     |                                  |
--|                     |      To_Next_Value               |
--|                     |__________________________________|
--|                     |                                  |
--|
procedure Get_Next_Valid_Record(Iterator : in out Iterator_Type)
   is
   begin

   loop
      --Ada.Text_Io.Put_Line("  Iterator.Hash_Table_Index" & Integer'Image(Iterator.Hash_Table_Index));

      exit when Iterator.Current_Node_Access /= null;
      Iterator.Hash_Table_Index := 1 + Iterator.Hash_Table_Index;
      exit when Iterator.Hash_Table_Index > Iterator.Number_Of_Buckets;
      Iterator.Current_Node_Access := Iterator.Hash_Table_Array_Access.all (Iterator.Hash_Table_Index);

   end loop;

end Get_Next_Valid_Record;
--|                     |__________________________________|
--|                     |                                  |
--|                     |      To_Next_Value               |
--|                     |__________________________________|
--|                     |                                  |
--|
procedure To_Next_Value(Iterator : in out Iterator_Type)
   is
   begin
   -- go  one step further
   Iterator.Current_Node_Access := Iterator.Current_Node_Access.Access_To_Next;
   -- get next valid record (if any)
   Get_Next_Valid_Record(Iterator => Iterator);
end To_Next_Value;
--|
--|                     |__________________________________|
--|                     |                                  |
--|                     |            Initialize            |
--|                     |__________________________________|
--|                     |                                  |
--|
procedure Initialize(Iterator           :    out Iterator_Type;
                     For_The_Hash_Table : in out Linked_Hash_Table_Type)
   is
   begin
   --   iterator can be used several times so init everything every time
   Iterator.Hash_Table_Index        := 0;
   Iterator.Current_Node_Access     := For_The_Hash_Table.Hash_Table_Array(Iterator.Hash_Table_Index);
   Iterator.Number_Of_Buckets       := For_The_Hash_Table.Number_Of_Buckets;
   Iterator.Hash_Table_Array_Access := new Hash_Table_Array_Type'(For_The_Hash_Table.Hash_Table_Array);
   -- seek  first non empty position in hash table  (if any)
   Get_Next_Valid_Record(Iterator => Iterator);

end Initialize;
--|
--|                     |__________________________________|
--|                     |                                  |
--|                     |         Initialize               |
--|                     |__________________________________|
--|                     |                                  |
--|
procedure Initialize(Hash_Table : in out Linked_Hash_Table_Type)
   is
   Dummy_Node_Access : Node_Access_Type;
   begin

   for Local_Bucket_Number in Hash_Table.Hash_Table_Array'Range loop
      while null /= Hash_Table.Hash_Table_Array(Local_Bucket_Number) loop

         Dummy_Node_Access := Hash_Table.Hash_Table_Array(Local_Bucket_Number);
```

```
            Hash_Table.Hash_Table_Array(Local_Bucket_Number) := Dummy_Node_Access.Access_To_Next;

            Free_Node(Dummy_Node_Access);
         end loop;
      end loop;
      Hash_Table.How_Many_Stored := 0;


      Statistics.Add_Calls    := 0;
      Statistics.Delete_Calls := 0;
      Statistics.Seek_Calls   := 0;
   end Initialize;
   --|
   --|                      |_____________________________|
   --|                      |                             |
   --|                      |       Print_Statistics      |
   --|                      |_____________________________|
   --|                      |                             |
   --|
   procedure Print_Statistics(Hash_Table : in     Linked_Hash_Table_Type)
      is
      --package My_Float_Io is new Ada.Text_Io.Float_Io(Float);
      begin
      Ada.Text_Io.Put_Line("How_Many_Stored" & Integer'Image(Hash_Table.How_Many_Stored));

      Ada.Text_Io.Put_Line(" Add_Calls" & Integer'Image(Statistics.Add_Calls)
                           & " Delete_Calls" & Integer'Image(Statistics.Delete_Calls)
                           & " Seek_Calls" & Integer'Image(Statistics.Seek_Calls));
   end Print_Statistics;
end Linked_Hash;
```