TD Numéro 12 (3 heures)

Manipulation des caractères et des chaînes de caractères.

Manipulation des caractères (à survoler rapidement car c'est du rappel !).

Le type `Character`, prédéfini dans le paquetage Standard, représente le jeu standard ISO 8-bits ISO 8859-1, appelé communément `Latin_1`. Il contient l'ensemble des caractères utilisés pour les entrées-sorties et comprend les différents caractères accentués en usage dans les langues européennes.

En Ada 95 deux paquetages facilitent la manipulation des caractères. Ce sont :
`Ada.Characters.Latin_1` et `Ada.Characters.Handling`

Le paquetage `Characters.Latin_1` contient la déclaration de constantes donnant des noms mnémoniques à la plupart des caractères, ce qui permet de s'y référer facilement, comme par exemple `Characters.Latin_1.Asterisk`. La classification des caractères et les noms de `Latin_1` sont donnés dans le polycopié « paquetages Ada » à consulter.

Le paquetage `Characters.Handling` contient un certain nombre de fonctions de classement et de fonctions de conversion pour les caractères (`Character`) et les chaînes (`String`). La spécification de `Characters.Handling` est donnée dans le polycopié « paquetages ».

En général, les noms de ces fonctions correspondent à leur signification en anglais.
Exemple (le caractère est-il un caractère dit de contrôle ?) :

```
function Is_Control (Item : in Character) return  Boolean ;
```

Il existe d'autres fonctions (du type : "le caractère est-il ?") voyons les toutes :

- `Is_Control` caractère de contrôle (intervalle 0..31)
- `Is_Graphic` caractère graphique (n'est pas un caractère de contrôle)
- `Is_Letter` lettre
- `Is_ Lower` lettre minuscule
- `Is_Upper` lettre majuscule
- `Is_Basic` caractère de base (pas de lettre accentuée)
- `Is_Digit` chiffre
- `Is_Decimal_Digit` chiffre décimal
- `Is_Hexadecimal_Digit` chiffre hexadécimal
- `Is_Alphanumeric` lettre ou chiffre
- `Is_Special` ni chiffre ni lettre ni caractère de contrôle

Exemple de fonctions de conversion (remarquez la surcharge !) :

```
function To_Lower (Item : in Character) return Character;
function To_Lower (Item: in String) return String;
```

Il existe aussi d'autres fonctions `To_Upper, To_Basic` etc. cf. polycopié « paquetages … »

Remarque : En Ada95 il existe également le type prédéfini `Wide_Character` correspondant au jeu ISO 10646. Ce type de caractères correspond au jeu ISO 16 bits BMP (Basic Multilingual Plane). Les 256 premiers littéraux de ce jeu correspondent à ceux du type `Character`. Ces 256 littéraux sont donc **partagés** ⇒ problème d'ambiguï té !

Manipulation des chaînes.

Rappel : Le type `String` a été présenté (voire critiqué !) dans le cours Ada. Résumons :

- Il est prédéfini dans le paquetage `Standard`. Voir polycopié paquetages Ada.
- C'est en fait un tableau non contraint de caractères, c'est pourquoi pour manipuler des variables de type `String` il faut d'abord contraindre les indices puis instancier.
- Les opérateurs associés au type `String` sont ceux permis pour les types tableaux à composants discrets (ce qui est le cas du type `Character`) donc on dispose des 6 opérateurs =, /=, <, >, <=, >= ainsi que de l'opérateur de concaténation &.
- Les attributs permettant les opérations sur les tableaux sont : `First`, `Last`, `Range` et `Length`. Ils sont donc applicables au type `String`.
- La saisie d'un `String` n'est pas évidente (vue dans le cours sur les Entrées/Sorties simple (5 bis)). Il était intéressant d'utiliser `Lire` de `P_E_Sortie` qui ajoute des espaces pour saturer la chaîne. Il est aussi très pratique de lire des `Unbounded_String` (cf. cours Ada). Nous les utiliserons.
- Les opérations sur les chaînes de caractères reviennent souvent à manipuler des tranches de tableaux de caractères. Il est possible d'affecter une tranche de tableau à une autre tranche de tableau. Il nous faut cependant connaître les intervalles de ces tranches. Ce n'est pas toujours le cas. Revoir l'exemple : comment affecter une chaîne à une autre chaîne si on ne connaît pas sa longueur ? Voir aussi, dans le cours, l'utilisation plus simple grâce à un `Unbounded_String`.

Remarque: Nous venons de voir qu'en Ada les chaînes de caractères sont des tableaux non contraints de caractères. Il n'en est pas de même selon les langages. En PASCAL une chaîne de caractères est un tableau d'octets dont l'élément en position 0 contient une valeur entière représentant le nombre de caractères (255 caractères maximum). En C++ et en C une chaîne de caractères est une suite d'octets se terminant par le caractère "\0" (Premier caractère de la table ASCII noté `NUL` en Ada dans le paquetage `Latin_1`).

La norme Ada95 a prévu de nouveaux paquetages fournissant des fonctionnalités (évitant de réinventer la roue !) pour la manipulation des chaînes. Ce sont :

```
Ada.Strings
Ada.Strings.Maps
Ada.Strings.Maps.Constants
Ada.Strings.Search
Ada.Strings.Fixed
Ada.Strings.Bounded
Ada.Strings.Unbounded;
Ada.Strings.Unbounded.Text_Io ;
```

Voir le polycopié « paquetages Ada ». Tous les A.4. (1, 2…,6) Nous allons les étudier.

A utiliser impérativement donc … à connaître.

Classification.

Le paquetage `Ada.Strings.Fixed` sert aux types chaînes ayant des tailles fixes (donc le type `String`). L'utilisateur a **la charge de gérer la longueur utile**.
Le paquetage `Ada.Strings.Bounded` sert aux types chaînes (type privé `Bounded_String`) ayant une longueur maximale, mais, dont seule une partie est utilisée (voir plus bas).
Le paquetage `Ada.Strings.Unbounded` traite des chaînes dites non bornées (type privé `Unbounded_String`) bien que nécessairement bornées dans les faits par `Integer'Last`.
Ces deux derniers types (contrairement aux `String`) « portent » avec eux leur longueur utile et celle-ci est **variable** avec le temps. Cette longueur utile est gérée automatiquement ce qui libère l'utilisateur. C'est une belle réponse aux archaï ques `String`. L'idée générale de ces trois paquetages est que les trois types de chaînes doivent, dans la mesure du possible, être traitées de la même façon dans les paquetages. On trouvera pas mal de surcharges.

Le paquetage parent `Ada.Strings` (voir la spécification dans le polycopié « paquetage Ada ») contient des constantes et des déclarations de types énumératifs concernant le contrôle des différentes options et des exceptions qui seront levées si quelque chose se passe mal. Ces déclarations sont donc accessibles aux 3 paquetages fils (vus ci dessus) qui nous intéressent ! On trouve 4 exceptions et 5 types (cf. le polycopié !) :

```
Length_Error,Pattern_Error,Index_Error,Translation_Error : exception;

type Alignment  is (Left, Rigth, Center);  -- gauche, droite, centrée
type Truncation is (Left, Right, Error); -- gauche, droite, erreur
type Membership is (Inside, Outside);  -- dedans, hors de
type Direction  is (Forward, Backward); -- en montant, en descendant
type Trim_End   is (Left, Right, Both); -- gauche, droite, les deux
```

Nous allons survoler ces trois paquetages fils « Fixed », « Bounded » et « Unbounded » en considérant leurs nombreuses opérations. Les spécifications complètes sont à voir dans le polycopié « paquetages Ada ». Les chaînes (non fixes) bornées et non bornées ont une borne inférieure égale à 1 ce qui n'est pas toujours le cas avec les chaînes fixes. En effet une variable de type `String` peut être déclarée (rappel) de la façon suivante :

```
Une_Chaine : String (11..20);
```

Cependant on n'envisagera pour notre compréhension que des « Fixed » ayant une borne inférieure égale à 1 c'est à dire avec un intervalle débutant toujours à 1 telle :

```
Une_Chaine : String (1..10);
```

Les chaînes fixes (`String`) et bornées (`Bounded_String`) peuvent poser un problème concernant leur capacité contrairement aux chaînes non bornées (`Unbounded_String`). Pour qu'une grande partie des fonctionnalités s'appliquent aux chaînes fixes et bornées on introduira la notion de **justification** et de **remplissage**.

Attention : toutes les fonctionnalités ne seront (faute de place et de temps) pas envisagées ici. Il est recommandé de comparer les paquetages (notamment les plus intéressants `Bounded` et `Unbounded`) pour noter leur différence et **imaginer d'autres exemples**.

Déclarations utiles pour les exemples à venir :

Après les 3 déclarations suivantes (hors les **with** !) :

Notez le s à Strings !

```
use Ada.Strings.Fixed;
use Ada.Strings.Bounded;
use Ada.Strings.Unbounded;
```

```
et après l'instanciation :
package Chaine_80 is new Generic_Bounded_Length(80);
use Chaine_80;
```

il est possible de créer des variables chaînes des différents types :

```
C    : Character;
S5   : String(1..5);
S10  : String(1..10);
S15  : String(1..15);

Bs,
Bs2  : Bounded_String;   -- vides a priori

Us,
Us2  : Unbounded_String; -- vides a priori

Lbs  : Length_Range; -- sous-type de Natural pour Bounded
Ls, I, J : Natural;
```

Survol des fonctionnalités des paquetages Fixed, Bounded et Unbounded.

Fonctionnalité de longueur de chaîne (`Bounded_String` et `Unbounded_String`).

Les longueurs des chaînes de type `Bounded_String` et `Unbounded_String` étant variables avec leur utilisation, il est nécessaire de pouvoir connaître la longueur **courante** (utilisée à l'instant) d'une chaîne. Cette longueur est donnée par la fonction `Length` :

```
Lbs := Length (Bs) ;
Ls := Length (Us) ;
```

Pour le type `String` cette fonction est inutile puisque l'on dispose de l'attribut `Length`. Mais la longueur ne varie jamais ! L'attribut donne la longueur maxi et **jamais** la longueur utile !

```
Ls := S5'Length   -- 5 évidemment
```

Fonctionnalité de conversion (`String`, `Bounded_String` et `Unbounded_String`).

Les paquetages "Bounded" et "Unbounded" fournissent **seulement** les 3 fonctions de conversion `To_Bounded_String`, `To_Unbounded_String` et `To_String`.

```
function To_Bounded_String(Source:in String;
                           Drop:in Truncation := Error) return Bounded_String;
function To_Unbounded_String (Source : in String) return Unbounded_String;
function To_String (Source : in Bounded_String)  return String;
function To_String (Source : in Unbounded_String) return String;
```

Ces fonctions permettent de convertir (respectivement) une chaîne de type `String` en chaîne de type `Bounded_String` ou `Unbounded_String` et inversement de convertir un `Bounded_String` ou un `Unbounded_String` en `String`. Le passage d'une chaîne de type `Bounded_String` en chaîne de type `Unbounded_String` se fait par l'intermédiaire d'une chaîne de type `String`.

```
Bs := To_Bounded_String("Une chaîne");
Us := To_Unbounded_String ("");
-- ici une chaîne vide c'est équivalent à Us := Null_Unbounded_String ;
Us := To_Unbounded_String (To_String (Bs));
```

Fonctionnalité de <u>concaténation</u> (`Bounded_String` et `Unbounded_String`).

Le paquetage `Bounded` fournit des fonctions et des procédures de concaténations : & et `Append`. La fonction & (opérateur surchargé) permet de créer une nouvelle chaîne avec des opérandes : caractère, chaîne fixe ou chaîne bornée. La fonction & lèvera l'exception `Length_Error` en cas de dépassement de la borne (valeur donnée à l'instanciation du paquetage). Les sous-programmes `Append` (procédures et fonctions) permettent des contrôles (ce que ne permet pas &). Ils ont un paramètre formel `Drop` (de type `Truncation` voir page 3) qui provoquera la perte des caractères en trop si sa valeur effective est `Left` ou `Right` ou lèvera l'exception `Length_Error` si sa valeur est `Error` (par défaut). Quelques exemples :

```
function Append (Left : in Bounded_String; Right : in Character;
                 Drop : in Truncation := Error)
                                        return Bounded_String;
procedure Append (Source : in out Bounded_String;
                  New_Item : in Bounded_String;
                  Drop     : in Truncation  := Error);
function "&" (Left : in Bounded_String; Right : in String)
                                          return Bounded_String;
```

Voyez s'il en manque ! Applications :

```
Bs := To_Bounded_String("Une première chaine" & ' '
                        & "Une deuxième chaine");
Bs := Append(Bs,' ' &
                   "Une troisième chaine",Right)); -- fonction
Append(Bs,' ' & "Une quatrième chaine", Left);     -- procédure
```

Imaginez d'autres exemples.

Les sous-programmes `Append` et les fonctions & existent aussi pour les chaînes non bornées (Unbounded) mais <u>sans le troisième paramètre</u> ou <u>sans levée d'exception</u> (puisque la capacité est sans importance, presque infinie !).

Les sous programmes `Append` sont absents dans le paquetage Fixed on comprend pourquoi ! Mais, rappel, la fonction & est prédéfinie pour le type `String` (cf. paquetage `Standard`) donc non déclarée dans le paquetage Fixed.

Fonctionnalité de sélection (`String`, `Bounded_String` et `Unbounded_String`).

Les paquetages Bounded et Unbounded fournissent la fonction `Element` et la procédure `Replace_Element`. La fonction `Element` renvoie **le** caractère situé en position `Index` dans la chaîne et la procédure `Replace_Element` remplace le caractère situé en position `Index` par le caractère `By`. Ceci était facilement réalisable pour les chaînes fixes avec un indice (ici, et c'est rare : avantage aux `String` !).

```
function Element (Source: in Bounded_String; Index : in Positive)
                                              return Character;

procedure Replace_Element (Source : in out Bounded_String;
                           Index  : in Positive;
                           By     : in Character);

C := Element (Us, I);          -- extrait le Ième élément
Replace_Element (Us, I ,'a');  -- remplace le Ième élément par 'a'
```

Les deux paquetages fournissent également la fonction `Slice` et les sous programmes `Replace_Slice`. La fonction `Slice` permet d'extraire (c'est une copie) une partie d'une chaîne entre les index `Low` et `High` (c'est une tranche de caractères **donc c'est un `String`**) .

La procédure et la fonction `Replace_Slice`, **peuvent** même insérer un **nombre supérieur ou inférieur** d'éléments qui ont été enlevés. Quelques exemples :

```
function Slice (Source : in Bounded_String; Low : in Positive;
                High   : in Natural) return String;

function Replace_Slice (Source: in Bounded_String;
                        Low   : in Positive;
                        High  : in Natural;
                        By    : in String;
                        Drop  : in Truncation := Error)
                                    return Bounded_String;

S(I..J) := Slice (Us, I, J); -- copie les éléments de I à J de Us
Bs2 :=  Replace_Slice(Bs, 2, 5, S(I..J), Error) ;
```

Attention ! **`String`**

La fonction `Slice` extrait les caractères compris entre I et J de Us (l'affectation avec un `String` doit être rigoureuse (rappel !) d'où le : `S(I..J)` et `Replace_Slice` supprime les caractères compris entre les positions 2 et 5 de Bs et les remplace par les caractères contenus dans la sous chaîne `S(I..J)`quelle que soit l'amplitude de `I..J` (s'il y a plus ou moins de 4 caractères c'est sans importance) Le paramètre de type `Truncation` (ici `Drop => Error`) gèrera le « débordement » **global** et non le remplacement local.

`Replace_Slice` existe aussi en procédure, exemple :

```
Replace_Slice (Us, 2, 5, S(I..J));
```

Pas de paramètre `Truncation` avec les `Unbounded_String` (évident !).

Fonctionnalité de comparaison (`Bounded_String` et `Unbounded_String`).

Les fonctions de comparaisons, (=, <, etc.) sont nécessaires dans les paquetages Bounded et Unbounded. Elle sont inutiles dans Fixed puisque ces fonctions sont prédéfinies pour le type `String`.

```
if Bs < Bs2
if Us2 <= Us  etc.
```

Fonctionnalité de recopie (`move` pour les Fixed vue dans un TD avec P.S.).

Autres fonctionnalités de sélection (`String`, `Bounded_String` et `Unbounded_String`)

La recherche d'une sous-chaîne `String` dans une chaîne est réalisée par les fonctions `Index`. Il existe des fonctions `Index` pour les trois types de chaînes. Les fonctions `Index` retournent l'indice du début de la sous-chaîne ou 0 si la sous-chaîne n'est pas trouvée. Le `String` recherché s'appelle le "patron".

```
function Index (Source : in Bounded_String ;
                Pattern : in String;
                Going  : in Direction:= Forward;
                Mapping : in Maps.Character_Mapping := Maps.Identity)
                          return  Natural;
```

Le patron est un String !

Exemples :

```
Bs  := To_Bounded_String ("IUT Aix-en-Provence");
Lbs := Index (Bs, "Aix") -- c'est 5
Lbs := Index (Bs, "aix") -- c'est 0
```

La direction de la recherche est donnée par un paramètre `Going` qui peut être `Forward` pour une recherche à partir du début (par défaut s'il est absent) ou `Backward` pour une recherche à partir de la fin.

```
Lbs := Index (Bs, "Pro" , Backward);  -- ici c'est 12
```

Le dernier paramètre `Mapping` est du type privé `Character_Mapping` déclaré dans le paquetage `Ada.Strings.Maps`. Une valeur de ce type (sauf pour la valeur « neutre » `Maps.Identity`) assouplit la « casse ». Les 3 valeurs standards (toutes terminées par `Map` !) sont définies dans `Ada.Strings.Maps.Constants`. Exemple :

`Lower_Case_Map` « associe » les caractères majuscules à leur correspondant minuscule et laisse les autres inchangés. Les autres sont :

`Upper_Case_Map` (mise en majuscules) et `Basic_Map` (suppression des accents)

```
I := Index (Bs, "toto", Forward, Lower_Case_Map);
```

Tous les caractères contenus dans Bs seront traités comme des caractères minuscules même si certains sont majuscules et seront comparés (4 par 4) avec la chaîne "toto". Si Bs contient "Toto" par exemple alors sa place dans Bs sera bien rendue par I.

Autre exemple :

```
I := Index(Bs,To_String(Bs2), Forward, Upper_Case_Map);
```

Il existe une autre version de la fonction `Index` cherchant la première occurrence d'un ensemble de caractères (`Character_Set`) qui sert alors de "patron" pour la recherche.

```
function Index(Source : in Bounded_String ;
               Set    : in Maps.Character_Set;
               Test   : in Membership:= Inside;
               Going  : in Direction := Forward)
                                  return  Natural;
```

Ce n'est plus un String !
C'est un **ensemble**

Elle utilise le type `Character_Set` déclaré aussi dans `Ada.Strings.Maps`. Il existe de nombreuses constantes de type « ensemble » définies dans `Ada.Strings.Maps.Constants`.
Ces constantes sont toutes terminées par `Set` !. Voir en A.4.6.

```
Control_Set     Upper_Set           Alphanumeric_Set
Graphic_Set     Basic_Set           Special_Set
Letter_Set      Decimal_Digit_Set   ISO_646_Set
Lower_Set       Hexadecimal_Digit_Set
```

Exemples :

```
   I := Index(Bs , Set => Lower_Set) ;
-- donne la position de la première lettre minuscule
```

Vous pouvez «construire » **votre propre ensemble** avec les fonctions `To_Set` (à partir d'une énumération sous la forme d'un `String`).

```
   I := Index(Bs , Set => To_Set("aeiouyAEIOUY") ;
```

Ainsi on trouvera la position (si elle existe) de la première voyelle non accentuée.

Deux autres paramètres optionnels vous permettent de chercher le premier caractère **en dehors de l'ensemble** (qui n'appartient pas à l'ensemble) et aussi de préciser la direction de recherche. Ainsi :

```
Bs:=  To_Bounded_String ("IUT Aix-en-Provence");
I := Index (Bs, Lower_Set, Outside, Forward);
-- le premier caractère non minuscule en montant est en position 1
I := Index (Bs, Lower_Set, Outside, Backward);
-- le premier caractère non minuscule en descendant est en position 12
```

Il est possible pour les trois types de chaînes de compter le nombre d'occurrences d'un patron `String` (ou d'un ensemble) dans une chaîne (quelconque) à l'aide des fonctions `Count`.

```
I:=  Count (Bs, "AB", Upper_Case_Map);
I:=  Count (Bs, To_String(Bs2), Upper_Case_Map);
I:=  Count (Bs, To_Set(",;!:."); -- comptage de la ponctuation!
```

On peut aussi chercher une « séquence » (ou plage) dont les caractères satisfont certaines propriétés (à préciser) avec la procédure `Find_Token`.

```
procedure Find_Token (Source : in Bounded_String;
                      Set    : in Maps.Character_Set;
                      Test   : in Membership;
                      First  : out Positive;
                      Last   : out Natural);
```

Notez pas de valeur par défaut ! Et notez que le zéro est possible !

Exemples :

```
Find_Token (Bs, Lower_Set, Inside, I, J);
Find_Token (Bs, Lower_Set, Outside, I, J);
Find_Token (Bs, To_Set("0123456789", Inside, I, J);
```

Premier exemple : si Bs contient les caractères "Comtesse Lovelace" I vaudra 2 et J vaudra 8 puisque la tranche `2..8` de Bs est la première dont l'ensemble des caractères sont minuscules. Le troisième paramètre de type `Membership` a pour valeur `Inside` (Contenu dans l'ensemble des minuscules).
Deuxième exemple: valeur du paramètre de type `Membership` est `Outside` (non contenu dans l'ensemble). I vaudra 1 et J vaudra 1 également. La première "tranche" non minuscule se réduit à l'intervalle 1..1.
Dans le troisième exemple I vaut 1 (un) et J vaut 0 (zéro) (aucun caractère chiffre rencontré : intervalle vide 1..0).

Fonctionnalités de conversion (`String`, `Bounded_String` et `Unbounded_String`).

Pour les trois types de chaînes les sous-programmes `Translate` permettent de convertir les chaînes notifiées par une correspondance de caractères. Il existe une forme procédurale et une forme fonction.

```
function Translate (Source  : in Bounded_String;
                    Mapping : in Maps.Character_Mapping)
                                 return Bounded_String;
procedure Translate (Source : in out Bounded_String;
                     Mapping : in Maps.Character_Mapping);

Bs2 := To_Bounded_String("ADA95");
Bs := Translate (Bs2,  Lower_Case_Map); -- "ada95";

Bs2 := To_Bounded_String("Prédéfini");
Bs := Translate (Bs2, Basic_Map);   -- "Predefini"
```

Fonctionnalités de transformation (`String`, `Bounded_String` et `Unbounded_String`).

Les sous-programmes de transformation (procédures et fonctions) sont fournis pour les trois types de chaînes. Il s'agit de `Insert`, `Delete` et `Overwrite`.

```
function Insert (Source   : in Bounded_String;
                 Before   : in Positive;
                 New_Item : in String;
                 Drop     : in Truncation := Error)
                                    return Bounded_String;
procedure Insert (Source   : in out Bounded_String;
                  Before   : in Positive;
                  New_Item : in String;
                  Drop     : in Truncation := Error);
function Overwrite (Source    : in Bounded_String;
                    Position  : in Positive;
                    New_Item  : in String;
                    Drop      : in Truncation := Error)
                                     return Bounded_String;
procedure Overwrite (Source    : in out Bounded_String;
                     Position  : in Positive;
                     New_Item  : in String;
                     Drop      : in Truncation := Error);
```


```
function Delete (Source  : in Bounded_String;
                 From    : in Positive;
                 Trough : in Natural)
                        return Bounded_String;
procedure Delete (Source  : in out Bounded_String;
                  From    : in Positive;
                  Through : in Natural);
```

Les sous-programmes `Insert` permettent **d'insérer** une sous-chaîne (attention `String`) dans une chaîne (quelconque) à partir d'une position indiquée dans la chaîne.

Les sous-programmes `Delete` permettent de supprimer une **plage** de caractères (intervalle) dans une chaîne.

Les sous-programmes `Overwrite` permettent de **remplacer** à partir d'une position une sous chaîne par une autre sous-chaîne (de même taille) `String`.

On retrouve le paramètre `Drop` pour les chaînes fixes et pour les `Bounded_String` avec les sous-programmes `Insert` et `Overwrite` pour gérer les débordements.

```
Insert(Bs,3,To_String(Bs2),Drop => Right);        -- coller
Overwrite (Bs, 3, To_String(Bs2),Drop =>  Right); -- couper-coller
Delete(Bs,2,4);                                   -- couper

Bs2 := Insert (Bs,3,To_String(Bs2), Drop => Error);
Bs2 := Overwrite (Bs,3,To_String(Bs2),Drop => Right);
Bs2 := Delete (Bs,2,4);
```

Fonctionnalité d'extraction (`String`, `Bounded_String` et `Unbounded_String`).

On trouve les sous-programmes (procédures et fonctions) `Trim`, `Head` et `Tail`.

Les 4 sous-programmes `Trim` permettent de supprimer des caractères particuliers (notamment l'espace dans les deux premières versions) à partir de l'une et/ou de l'autre extrémité. Il est possible de préciser l'ensemble des caractères à supprimer (cf. deux dernières versions). `Trim` comme son nom l'indique embellit (ou nettoie) la chaîne. Voir le polycopié .

Deux exemples :

```
function Trim (Source : in Bounded_Strig;
                 Side : in Trim_End)
                    return Bounded_String;

procedure Trim (Source : in out Bounded_String;
                Left   : in Maps.Character_Set;
                Right  : in Maps.Character_Set);


Bs := Trim (Bs , Side => Both) ;
-- supprime les blancs en tête et en queue
Trim (Bs , Left => Decimal_Digit_Set , Right => Null_Set);
-- supprime les chiffres en tête mais rien en queue
```

Les sous-programmes `Head` et `Tail` extraient en tête ou en queue d'une chaîne une sous chaîne comprenant un nombre donné de caractères (s'il en manque ils ajoutent le caractère de « bourrage » `Pad`). `Drop` n'existe pas pour les `Unbounded_String` (évidemmment).

```
function Head (Source : in Bounded_String;
               Count  : in Natural;
               Pad    : in Character  := Space;
               Drop   : in Truncation := Error)
                              return Bounded_String;
procedure Head (Source : in out Bounded_String;
                Count  : in Natural;
                Pad    : in Character  := Space;
                Drop   : in Truncation := Error);

function Tail (Source : in Bounded_String;
               Count  : in Natural;
               Pad    : in Character  := Space;
               Drop   : in Truncation := Error)
                                 return Bounded_String;

Bs := To_Bounded_String("IUT Aix-en-Provence");
Bs2 := Head (Bs,3); -- "IUT"
Bs2 := Tail (Bs,6); -- "ovence"

Bs := To_Bounded_String("Ada");
Bs2 := Head (Bs,8,'*',Left); -- "Ada*****"
```

Fonctionnalités de <u>création de chaînes</u> (`String`, `Bounded_String` et `Unbounded_String`)

Des surcharges appropriées de l'opérateur multiplicatif "*" créent des chaînes qui sont des **duplications** de caractères ou de chaînes. Voir polycopié. Exemples :

```
Bs := 2 * "Chaine_1";      -- Chaine_1Chaine_1
S15(1..9):= 3 * "Ada";     -- AdaAdaAda
```

L'opérateur **<u>n'est pas commutatif</u>** ! Le premier paramètre est le facteur multiplicatif !
Il existe des fonctions `Replicate` pour les Bounded avec un paramètre optionnel `Drop`. Analogues à l'opérateur "*" mais permettant de gérer la « troncature ».

Nous terminerons notre survol des fonctionnalités de manipulations de chaînes en passant en revue les diverses exceptions :

- `Length_Error`. Le résultat ne correspond pas à la taille.
- `Pattern_Error`. La sous-chaîne `String` ("patron") est vide (pour `Index` ou `Count`).
- `Index_Error`. L'indice est en dehors de la chaîne (pour `Element`, `Replace_Element`, `Slice` ou `Replace_Slice`).
- `Translation_Error`. Les paramètres de `To_Mapping` sont incorrects.

Tableau récapitulatif des procédures et fonctions

Nom	Fonction	Procédure	Fixed	Bounded	Unbounded	Page
Length	Oui	Non	Non	Oui	Oui	4
To_String	Oui	Non	Non	Oui	Oui	4
To_Bounded_String	Oui	Non	Non	Oui	Non	4
To_Unbounded_String	Oui	Non	Non	Non	Oui	4
&	Oui	Non	Oui	Oui	Oui	5
Append	Oui	Oui	Non	Oui	Oui	5
Element	Oui	Non	Non	Oui	Oui	6
Replace_Element	Non	Oui	Non	Oui	Oui	6
Slice	Oui	Non	Oui *	Oui	Oui	6
Replace_Slice	Oui	Oui	Oui	Oui	Oui	6
<, >, <=, >=	Oui	Non	Oui	Oui	Oui	7
Move	Non	Oui	Oui	Non	Non	7
Index	Oui	Non	Oui **	Oui	Oui	7 et 8
Count	Oui	Non	Oui **	Oui	Oui	8 et 9
Find_Token	Non	Oui	Oui	Oui	Oui	9
Translate	Oui	Oui	Oui	Oui	Oui	9
Insert	Oui	Oui	Oui **	Oui	Oui	10
Overwrite	Oui	Oui	Oui **	Oui	Oui	10
Delete	Oui	Oui	Oui	Oui	Oui	10
Trim	Oui	Oui	Oui	Oui	Oui	10
Head, Tail	Oui	Oui	Oui	Oui	Oui	11
"*"	Oui	Non	Oui	Oui	Oui	12

* Non en ce qui concerne le résultat ou la cible mais Oui si l'on considère qu'un paramètre ("patron") est concerné.

** Oui en ce qui concerne la cible et aussi le paramètre "patron" !

Les entrées-sorties de chaînes de caractères

Entrées/Sorties pour les chaînes fixes (`String`).

En ce qui concerne les E/S pour les chaînes fixes, reportez vous au cours Ada n° 5 chapitre « lecture et écriture de variables de type `String` » avec le paquetage `Ada.Text_Io`.

Entrées/Sorties pour les chaînes non bornées (`Unbounded_String`).

Pour les `Unbounded_String` il **existe un paquetage** `Ada.Strings.Unbounded.Text_Io` (Voir Paquetages Ada). Il contient : Deux **fonctions** `Get_Line` pour la lecture, Deux procédures `Put` et deux procédures `Put_Line` pour l'écriture.

Entrées/Sorties pour les chaînes bornées (`Bounded_String`).

Pour les `Bounded_String` **il n'existe pas de paquetage** « officiel » pour les E/S. Nous allons en créer un ! Rappel : le paquetage `Ada.Strings.Bounded` contient un paquetage générique `Generic_Bounded_Length` qu'il est nécessaire d'instancier pour créer des objets de type `Bounded_String`. Exemple :

```
Package P_Bounded_80 is new
                      Ada.Strings.Bounded.Generic_Bounded_Length(80) ;

Bs1, Bs2 : P_Bounded_80.Bounded_String;
```

Nous allons créer un paquetage pour la lecture et l'écriture de ces variables. Ce paquetage sera générique et aura comme paramètre formel de généricité le paquetage noté `P_Bounded` future instanciation de `Ada.Strings.Bounded.Generic_Bounded`.

```
with Ada.Strings.Bounded, Ada.Text_Io;
generic
  with package P_Bounded is new
                      Ada.Strings.Bounded.Generic_Bounded_Length(<>);
package P_Bounded_Io is
   function  Get_Line return P_Bounded.Bounded_String;
   function  Get_Line (File : Ada.Text_Io.File_Type)
                       return P_Bounded.Bounded_String;
   procedure Put_Line (Item : in P_Bounded.Bounded_String);
   procedure Put_Line (File : Ada.Text_Io.File_Type;
                       Item : in P_Bounded.Bounded_String);
   procedure Put (Item : in P_Bounded.Bounded_String);
   procedure Put (File : Ada.Text_Io.File_Type;
                  Item : in P_Bounded.Bounded_String);
end P_Bounded_Io;
```

Analogue au paquetage Unbounded

Une fois le corps du paquetage réalisé, nous devrons instancier ce paquetage avec comme paramètre réel le vrai paquetage instancié du paquetage `Generic_Bounded_Length`. Exemple :

```
package P_Bounded_80_Io is new P_Bounded_Io(P_Bounded_80) ;
use P_Bounded_80_Io;
```

TP 12A (1 heure)

Objectif : Etude **pratique** du paquetage **`Ada.Strings.Bounded`**. Mise en œuvre des diverses fonctionnalités fournies par le paquetage générique **`Generic_Bounded_Length`**. Utilisation et test des diverses fonctions et des procédures. La découverte des possibilités des paquetages (ici `Bounded`) nécessite de mettre en pratique les notions parcourues en 3 heures. Hélas le temps imparti au TP (ici 1 heure) n'est pas suffisant et il **sera impératif de finir ce TP** dans la semaine (du courage !).

Méthode : (Voir schéma page 15)**.** A partir du paquetage **`Ada.Strings.Bounded`**, qui contient le paquetage générique **`Generic_Bounded_Length`**, créez le paquetage appelé **`P_Bounded_80`** par instanciation de ce paquetage générique (Longueur maxima des chaînes égale à 80 caractères). Vu déjà dans le TD précédent. Le paquetage **`P_Bounded_80`** fournit alors les diverses fonctionnalités des chaînes **`Bounded_String`** que nous allons manipuler.

Détails du TP :

- Créez un répertoire tp12A. Copiez les paquetages fournis.
- Voyez la création des paquetages **`P_Bounded_80`** et **`P_Bounded_80_Io ;`** en listant les « deux petits fichiers » `p_bounded_80.ads` et `p_bounded_80_Io.ads`. Voyez aussi le paquetage **`P_Bounded_Io`** cf. fichiers : `p_bounded_io.ads` et `p_bounded_io.adb`
- Voyez le fichier **`ts_bounded.adb.squ.`** Renommez ce fichier **`ts_bounded.adb.`** Ce fichier contient le squelette de la procédure **`Ts_Bounded`** dans laquelle nous allons tester (ie. essayer) l'ensemble des fonctionnalités mises à notre disposition par les paquetages **`P_Bounded_80`** et **`P_Bounded_80_Io`**. A réaliser et …. à rendre !
- Seules, les fonctions **`Length`**, **`To_String`** et **`To_Bounded_String`** sont « préparées ». Il vous faut procéder aux tests des autres fonctionnalités indiquées en commentaires en vous inspirant des premiers tests déjà écrits. **Imaginez ensuite d'autres tests**.

Vous remarquerez que de nombreuses saisies (à partir du clavier) seront nécessaires lors de l'exécution de votre programme. De plus vous serez amenés à faire de nombreuses exécutions pour contrôler votre programme au fur et mesure que vous rajouterez de nouveaux tests.
Il sera donc **plus astucieux** de placer les données **définitivement en une seule fois** dans un fichier (fichier **`bounded.in`**) et de rediriger ces données en entrée par la commande habituelle

```
Duo :~>./ts_bounded  < bounded.in
```

Il n'est plus nécessaire dans ce cas de laisser les instructions d'affichage du type :

```
Put (″Saisie de la première chaîne pour tester To_String ″ ) ;
```

Lors de l'exécution du programme de test on pourra aussi rediriger les sorties dans un fichier (**`bounded.out`**) par la commande complète :

```
Duo :~>ts_bounded  <bounded.in   >bounded.out
```

Afin de s'assurer du bon fonctionnement du programme de tests des fonctionnalités on vérifiera que le fichier **`bounded.out`** est identique aux espérances en fabriquant le fichier **`bounded.ora`** et en comparant les deux fichiers. Les deux fichiers **`bounded.in`** et **`.ora`** fournis concernent **notre** fichier de test Ts_Bounded et **non le vôtre** ! C'est un exemple !

On peut aussi concevoir l'algorithme avec des lectures fichier puisque les E/S le permettent !

TP 12 Partie I