Enseignant : M. C. PIOMBO

# TRAVAUX PRATIQUES
# Série 6

---

## Package

NB : Vous devez impérativement utiliser les méthodes de programmation structurée pour développer ces programmes (structure de contrôle, procédure, fonction).

**Exercice 1** : Analyse lexicale

On se propose d' analyser "lexicalement" une phrase saisie par l'instruction ADA suivante:

Get_line(s,n); où s est de type STRING(1..400) et n de type natural

On rappelle que n contient le nombre de caractères réellement saisis et s la chaîne de caractères.

Notre analyse lexicale consiste à :

- isoler chacun des "mots" de la chaîne (suite de caractères non blancs). Ce traitement retourne une liste dont chaque élément est une chaîne de caractères contenant un mot de la phrase saisie.
- puis, afficher la nature de chacun des mots.

  Il existe 3 natures de mot :

  - un entier qui est une chaîne dont chaque caractère appartient à ['0'..'9'];
  - un réel qui est une chaîne dont chaque caractère appartient à ['0'..'9'] et qui contient un seul caractère '.' (point);
  - une chaîne qui est une chaîne de caractères qui n'est ni un entier, ni un réel.

    **NB :** Chaque mot de la phrase sera constitué de 30 caractères au maximum

    Une phrase ne pourra contenir plus de 100 mots et pourra être vide.

    Un mot constitué d'un seul point (caractère '.') sera considéré comme une chaîne.

Exemple d'analyse lexicale : "exemple 123 entier 12.3 réel x12"

```
CHAINE : exemple
ENTIER : 123
CHAINE : entier
REEL   : 12.3
CHAINE : réel
CHAINE : x12
```

Affichage écran obtenu après analyse

***Travail demandé :***

écrire un programme ADA complet qui réalise et affiche le résultat de l'analyse comme présenté ci-dessus. C'est à dire pour chacun des mots de la phrase saisie, on doit connaître son type lexical et la valeur du mot.

**NB :** On suppose qu'un seul caractère blanc sépare chaque mot, et qu'il n'y a pas de blanc en début ou en fin de phrase

De l'analyse de l'application à réaliser on a déduit les points suivants :

***Structures de données associées à la liste de mots d'une phrase :***

| | |
|---|---|
| Liste_Mot = Elements + NbElements | ** liste de mots d'une phrase |
| Mot = Val + NB | ** Mot d'une phrase |
| Val : Chaîne de 30 caractères | ** caractères formant le mot (valeur du mot) |
| NB : entiers positif ou nul | ** nombre de caractères d'un mot |
| NbElements : entiers positif ou nul | ** nombre de mots dans une phrase |
| Elements : tableau de 100 Mot | ** ensemble des mots d'une phrase |

***Prototypes des Fonctions et procédure de bases :***

```
-- fonction qui retourne le nombre de mot d'une phrase
function Nbr_Mot(L : Liste_Mot) return Natural;

-- Fonction qui teste si un mot est un entier
function Isint(S:Mot) return Boolean;

-- Fonction qui teste si un mot est un reel
function Isfloat(S:Mot) return Boolean;

-- fonction qui retourne le Ième mot de la liste L
function Lire_Mot( I : Natural; L : Liste_Mot) return Mot;

-- fonction qui retourne la valeur (Val) du Ième mot de la liste L
function Lire_Val_Mot( I : Natural; L : Liste_Mot) return string;

-- Procedure qui isole les mots d'une Phrase S pour les stocker dans la liste L
procedure Isoler(S: String; L:in out Liste_Mot);
```

***Gestion des erreurs :***

```
-- exception gérant le fait qu' un mot d'une phrase
-- puisse comporter plus de 30 caractères
   mot_long : exception;
```

***L'application sera réalisée autour d'un package « Analyse_lexicale » qui contiendra les algorithmes codés en ADA des différentes fonctions et procédure présentées ci-dessus ainsi que les types et exception associés aux structures de données vues également plus haut dans le texte. Ces types devront être privés au package afin de répondre au critère d'encapsulation des données.***

***Un programme principal permettra d'obtenir le résultat demandé en mettant en œuvre le package « Analyse_lexicale ».***

***Utiliser plusieurs jeux de tests afin de valider vos algorithmes.***

Vous devrez fournir les listing des fichiers sources qui devront être clairs et documentés tout en respectant les règles de programmation structurée.

# La Généricité

**Exercice 2** : Une File

On se propose de construire un paquetage de gestion d'une file données. Le type File est bâti sur la structure de données suivantes :

**Type_File = Buffer + Compteur + Tete + Queue**
**Buffer : tableau de N données**
**Compteur : 0..N ** entier compris entre 0 et N**
**Tete, Queue : 0..N-1 ** entier compris entre 0 et N-1**

**<u>Les fonctions et procédures de manipulation de la file sont :</u>**

**Fonction** Créer_File : Type_File
** Retourne une nouvelle file vide

**Procédure** Ajouter( Valeur : données ; @Une_File : Type_File) ** @ = passage par Référence
** Ajoute une valeur en queue de la file Une_File
** ne peut être utilisée lorsque la file est pleine. Dans ce cas l'exception File_Pleine_erreur est levée

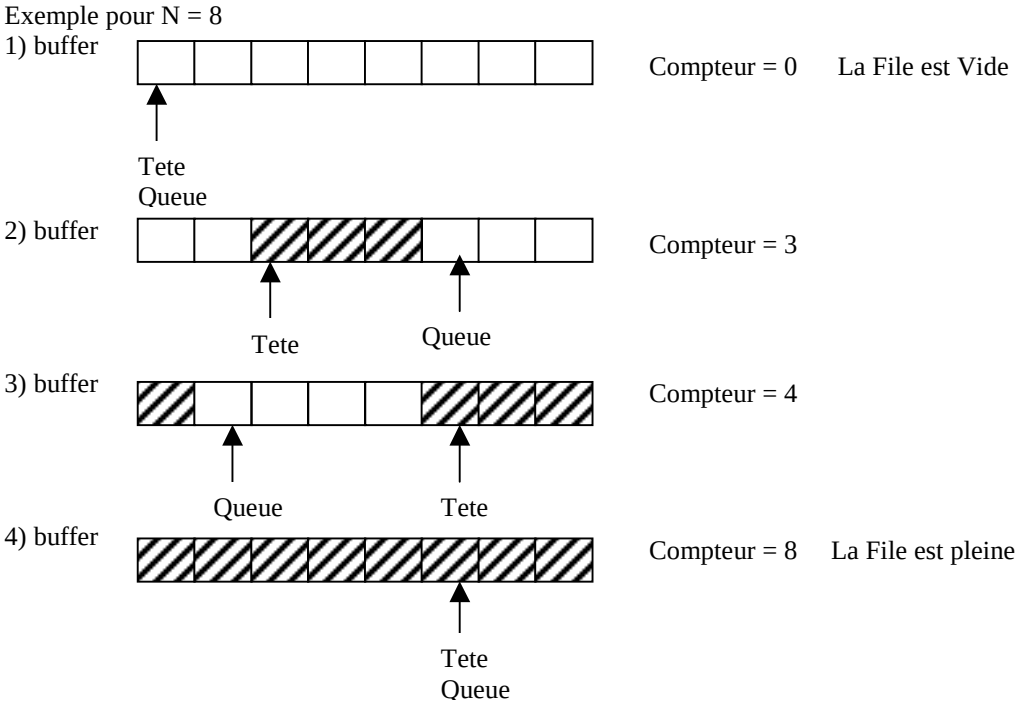
**Procédure** Retirer(@Valeur : données ; @Une_File : Type_File)
** retire de la file Une_File la valeur de Tête.
** ne peut être utilisée lorsque la file est vide. Dans ce cas l'exception File_Vide_erreur est levée

**Fonction** File_Vide(Une_File : Type_File) : Booléen
** retourne vrai si la file Une_file est vide, faux sinon

**Fonction** File_Pleine(Une_File : Type_File) : Booléen
** retourne vrai si la file Une_file est pleine, faux sinon

**Principe de fonctionnement de cette file:**

On range les valeurs des données dans un tableau de N éléments qui est géré de façon circulaire à l'aide des index **Tete** et **Queue**. La file de valeurs commence en Tete et est stockée de façon contiguë dans les places suivant Tete.
L'index Queue indique la première place libre en fin de la file de valeurs.
On ajoute donc en Queue et on retire en Tete.




Réaliser un programme qui mette en œuvre cette file de données en utilisant le principe de la généricité pour le paquetage de gestion de la file afin de pouvoir manipuler différents type de données.