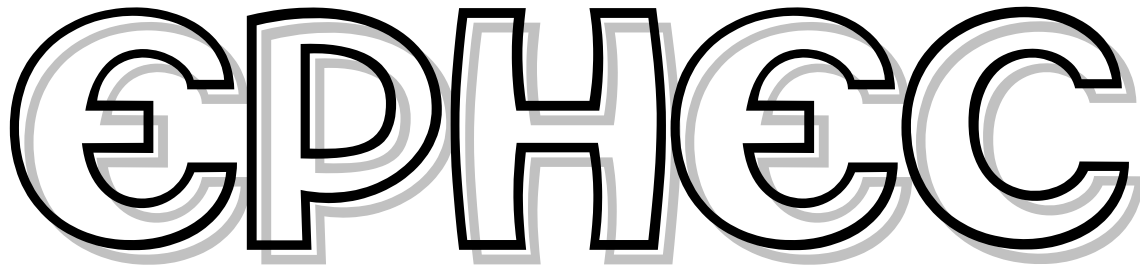


ÉCOLE PRATIQUE DES HAUTES ÉTUDES COMMERCIALES



INSTITUT D'ENSEIGNEMENT SUPÉRIEUR DE TYPE COURT

## PRINCIPES DE PROGRAMMATION

*ÉLÉMENTS D'ALGORITHMIQUE*

*PARTIE II : OUTILS DU RAISONNEMENT*



## TABLE DES MATIÈRES

1. INTRODUCTION .....	7
2. LES DONNÉES (I).....	9
2.1. LES TYPES DE DONNÉES.....	9
A) DÉFINITION .....	9
B) VALEURS LITTÉRALES.....	10
C) CONSTANTES.....	10
D) VARIABLES .....	11
E) EXPRESSIONS .....	12
2.2. LES TYPES SCALAIRES.....	13
A) LE TYPE LOGIQUE .....	14
B) LE TYPE ENTIER.....	16
C) LE TYPE CARACTÈRE .....	20
D) LE TYPE RÉEL.....	22
E) LE TYPE TEXTE .....	24
F) CONCRÈTEMENT ... ..	26
2.3. QUESTIONS & EXERCICES.....	28
2.4. LES TYPES STRUCTURÉS (I) .....	29
A) LES ENREGISTREMENTS.....	29
B) LES TABLEAUX.....	29
3. LES INSTRUCTIONS.....	31
3.1. LES INSTRUCTIONS D'ACTION.....	31
A) L'AFFECTATION.....	31
B) LES ENTRÉES/SORTIES .....	32
C) LA SÉQUENCE.....	32
D) CONCRÈTEMENT ... ..	33
E) LA PROCÉDURE .....	36
F) CONCRÈTEMENT ... ..	38
3.2. LES INSTRUCTIONS DE CONTRÔLE .....	42
A) LES CONDITIONS.....	42
B) LES STRUCTURES CONDITIONNELLES ET ALTERNATIVES.....	44
- Si ... Alors ... [ Sinon ... ] .....	44
- concrètement ... ..	48
C) STRUCTURES ALTERNATIVES MULTIPLES.....	51
- SelonQue ... valué.....	51
- SelonQue ... conditionné.....	53
- concrètement .....	54
D) EXEMPLE RÉCAPITULATIF COMMENTÉ .....	57
E) LES TABLES DE DÉCISION .....	66
- généralités .....	66
- construction .....	66
- alternatives .....	67
- simplifications.....	67
- un exemple .....	70
- un autre exemple.....	72
- concrètement .....	73

F) EXERCICES .....	75
G) LES STRUCTURES RÉPÉTITIVES (BOUCLES) .....	77
- TantQue .....	77
- Répéter ... Jusque .....	80
- comment choisir ? .....	82
- exemple récapitulatif .....	83
- d'autres (types de) boucles .....	84
- concrètement ... ..	85
H) EXERCICES .....	88
4. OUTILS DE GÉNÉRALISATION : PROCÉDURES ET FONCTIONS .....	91
4.1. PROCÉDURES .....	92
A) GÉNÉRALITÉS .....	92
B) PLUS D'INDÉPENDANCE : LES ÉTATS .....	93
C) LA PORTÉE DES VARIABLES ET DES PROCÉDURES .....	95
4.2. FONCTIONS .....	96
A) GÉNÉRALITÉS : RAPPEL SUR LA NOTION DE VALEUR .....	96
B) LA FONCTION : GÉNÉRALISATION DE LA NOTION DE VALEUR .....	97
4.3. NOTION DE PARAMÈTRE .....	101
4.4. PROCÉDURES PARAMÉTRÉES .....	104
A) ... SANS MODIFICATION DE DONNÉES .....	105
B) ... AVEC MODIFICATION DE DONNÉES .....	106
4.5. SOUPLESSE ET ABSTRACTION : LA SURCHARGE .....	108
4.6. CONCRÈTEMENT ... ..	110
4.7. EXERCICES .....	114
5. LES DONNÉES (II) .....	117
5.1. LIMITE DES DONNÉES SCALAIRES .....	117
5.2. LA STRUCTURE D'ENREGISTREMENT .....	121
A) PREMIER CONTACT .....	121
B) L'OPÉRATEUR D'ACCÈS "POINT" (.) .....	123
C) DÉCLARATIONS .....	123
D) LE MOT ENREGISTREMENT N'EST PAS UN IDENTIFICATEUR DE TYPE ! .....	124
E) LES TYPES ABSTRAITS (ADT & API), OUTILS D'ABSTRACTION .....	125
F) STRUCTURES ET SOUS-STRUCTURES ... ..	128
G) SUCRE SYNTAXIQUE .....	129
H) API : PROCÉDURES OU FONCTIONS ? .....	131
I) CONCRÈTEMENT .....	132
J) EXERCICES .....	135
5.3. LA STRUCTURE DE TABLEAU .....	137
A) PREMIER CONTACT .....	137
B) L'OPÉRATEUR D'ACCÈS [ ] .....	138
C) DÉCLARATIONS ET NOTATIONS .....	139
D) LE MOT TABLEAU N'EST PAS UN IDENTIFICATEUR DE TYPE ! .....	139
E) ITÉRATEURS BASIQUES POUR UN TABLEAU À UNE DIMENSION .....	141
F) L'ITÉRATEUR "POUR" .....	142
G) ITÉRATEURS POUR LA VALIDATION/VÉRIFICATION D'UN TABLEAU .....	145
H) TYPE TABLEAU ABSTRAIT : ADT & API .....	147
I) TAILLE PHYSIQUE ET TAILLE LOGIQUE D'UN TABLEAU PARTIEL .....	150
J) ITÉRATEURS (BOUCLES) IMBRIQUÉS .....	152

K) TABLEAUX COUPLÉS EXPLOITÉS "EN PARALLÈLE" .....	153
L) CONCRÈTEMENT ... ..	154
M) EXERCICES .....	156
5.4. STRUCTURES MIXTES .....	161
A) INTRODUCTION .....	161
B) RENFORCEMENT DE LA NOTION DE VALEUR .....	162
C) TABLEAUX PARTIELS .....	163
D) PETITE API BASIQUE POUR LA GESTION DE TABLEAUX PARTIELS .....	164
E) EXERCICES .....	167
5.5. ALGORITHMES CLASSIQUES SUR LES STRUCTURES AVEC TABLEAUX .....	169
A) TABLEAUX À CORRESPONDANCE DIRECTE .....	169
B) RECHERCHE(S) DANS UN TABLEAU NON ORDONNÉ .....	169
C) ORDONNANCEMENT (TRI) D'UN TABLEAU .....	175
- le tri 'bulle' .....	175
- le tri 'par sélection' .....	178
- variations sur le thème du tri .....	180
D) RECHERCHE DANS UN TABLEAU ORDONNÉ (TRIE) .....	184
- recherche séquentielle .....	184
- recherche dichotomique .....	185
E) TRIER OU INDEXER ? .....	188
F) GÉNÉRICITÉ DU TRI, DE L'INDEXATION ET DE LA RECHERCHE .....	192
G) NOTION DE FUSION .....	195
H) EXERCICES .....	198
5.6. POUR CONCLURE (?) : EXEMPLES .....	201
A) INTRODUCTION .....	201
B) EXEMPLE(S) COMPLET(S) .....	205
6. EXERCICES RÉCAPITULATIFS .....	211
7. EXAMENS DES ANNÉES PRÉCÉDENTES .....	225
SOURCES .....	237



## 1. INTRODUCTION

Nous allons à présent examiner plus en détail les différentes composantes d'un langage 'pseudo-code', pour établir les principes généraux d'une algorithmique :

- impérative sur base d'actions (donc d'instructions) et d'états (donc de valeurs),
- procédurale offrant des outils structurels du raisonnement, tels les procédures et les fonctions, qui fournissent une expressivité maximale, un bon niveau d'abstraction et une réutilisabilité maximale (modularité)
- itérative proposant les outils nécessaires à l'adaptabilité et à l'autonomie des algorithmes-automates, tels que diverses formes d'instructions conditionnelles, alternatives et de boucles.

Cependant, comme on a pu s'en rendre compte dans la première partie, du point de vue opérationnel, une telle algorithmique ne serait rien<sup>1</sup> sans les données concrètes permettant la mémorisation et la détermination d'états-valeurs, et conduisant donc à la programmation d'un ordinateur (le 'niveau générationnel' de langage utilisé n'ayant ici aucune importance particulière : depuis le langage-machine jusqu'aux langages-objet)

Les données ne seront cependant évoquées ici que de manière assez basique, réservant au cours d'Organisation et Structure des Données<sup>2</sup> leur étude plus approfondie ...

Après l'introduction de la notion capitale de type de données, les différents types scalaires seront examinés (types Logique, numériques, Caractère et Texte) et ce en termes de domaines de valeurs, d'opérateurs et de propriétés

Ensuite, deux structures essentielles, indispensables et parfaitement complémentaires, sans lesquelles l'automatisation de problèmes complexes serait certainement impossible, seront abordées : la structure inhomogène d'Enregistrement et la structure homogène de Tableau ainsi que différentes manières de les associer

Grâce à ces structures, différents algorithmes fondamentaux seront enfin étudiés, dont notamment la recherche, le tri et l'indexation ... et enfin la notion de fusion

<sup>1</sup> un ouvrage majeur de référence reste celui de N. Wirth (un des 'inventeurs' d'une algorithmique structurée basée sur le pseudo-code et le créateur – entre autres - du langage de programmation Pascal) : Algorithms + Data Structures = Programs. Prentice-Hall. ISBN 978-0-13-022418-7. 1976

<sup>2</sup> au second semestre ...





## 2. LES DONNÉES (I)

Comme on a pu le découvrir dans la première partie de ce texte, on ne sait faire de l'algorithmique 'concrète' (de la programmation) qu'en faisant appel aux données : ce sont elles en effet qui constituent les 'états' de l'algorithme (à partir desquels on peut déterminer - grâce aux opérateurs de comparaison - les états logiques vrai/faux qui interviennent dans les instructions de contrôle et donnent à l'ordinateur-automate son comportement adaptable).

### 2.1. LES TYPES DE DONNÉES

#### a) DÉFINITION

Du point de vue strictement algorithmique (qui n'a pas besoin d'avoir recours au concept d'ordinateur, c.-à-d. à une machine physique), les types de données sont une notion qui présente deux aspects essentiels :

- un ensemble de valeurs (ensemble au sens mathématique du terme, donc constitué de valeurs distinctes et ordonnées) ; cet ensemble est appelé domaine du type

*exemples*

- le domaine des nombres entiers relatifs pourrait être définis par la relation suivante :  

$$\dots -5 < -4 < -3 < -2 < -1 < 0 < 1 < 2 < 3 < 4 < 5 \dots$$
- le domaine des caractères représentables sur un ordinateur a pour caractéristique (compte tenu de la technique de codage utilisée):  

$$\dots 'A' < 'B' < 'C' < 'D' \dots$$

- un ensemble d'opérateurs internes au domaine (c.-à-d. qui, à partir d'une ou plusieurs valeurs du domaine, déterminent une autre valeur de ce même domaine)

*exemple*

- sur le domaine des nombres entiers l'addition, la soustraction, la multiplication, le quotient et le reste sont de tels opérateurs (leurs résultats sont entiers)

Notons dès à présent qu'il sera donc important de disposer également d'un ensemble d'opérateurs relationnels (de comparaison) permettant – à partir de deux valeurs d'un même domaine (d'un même type, donc) – de déterminer l'ordre qui existe entre ces valeurs (=, <, >, ≠, ≤, ≥)

*exemples*

- des expressions de comparaison telles que  $3 < 5$ ,  $-1 > 0$ ,  $'C' = 'A'$  sont en réalité des 'questions' basées sur l'ordre existant entre les données, et fournissant comme réponse un état logique vrai/faux

Du point de vue de la programmation (application de l'algorithmique en informatique, sur un ordinateur, plus précisément), les types de données présentent deux caractéristiques supplémentaires qui dépendent entièrement de l'implémentation (architecture interne, système d'exploitation, compilateur, interpréteur ...)

- une représentation physique finie (c.-à-d. limitée) des valeurs sur un ensemble d'octets, ce qui aura un impact essentiel sur le domaine des valeurs représentables
- une méthode de codage (c.-à-d. de transformation/représentation de l'information sous forme binaire, habituellement symbolisée par les valeurs 1 et 0)

b) VALEURS LITTÉRALES

Difficile de définir ce qu'est une valeur : *une information concrète reconnaissable directement, 'à vue', 'littéralement' ?*

Pour que chaque type de données possède son propre ensemble de valeurs sans qu'il y ait d'ambiguïté, il faut des règles syntaxiques d'écriture très strictes des valeurs.

Ainsi : 256, -333 sont des valeurs littérales numériques 'entières' ... alors que ...

256.01, -333.085 sont des valeurs littérales numériques 'décimales' (mais on préférera les qualifier de 'réelles') ...

'A' ou 'w' sont des valeurs littérales 'caractères' ... (on les place entre apostrophes)

"hello world !" est une valeur littérale 'texte' (on la place entre guillemets)

!!! attention !!! **1** est un entier alors que '**1**' est un caractère ... et "**1**" est un texte; avec le premier on sait faire des calculs, mais pas avec le second ni le troisième ... ce sont habituellement les possibilités d'utilisation qui déterminent si une valeur est numérique ou non : dans 1348 LLN, 1348 n'est pas (vraiment) numérique (on n'additionne pas des codes postaux); même remarque si 010335577 est un numéro de téléphone ...

c) CONSTANTES

Les constantes sont des valeurs littérales qui jouent un rôle particulier dans l'algorithme et ne sont pertinentes qu'en fonction du problème posé ...

par exemple,

- on peut imaginer dans RobotProg que la taille du terrain soit fixe - en nombre de cases - et nécessite pour ce faire la mémorisation de deux valeurs littérales : le nombre de cases selon l'axe x et le nombre de cases selon l'axe y ...
- de même, l'algorithme du Peleur a mis en évidence deux valeurs littérales particulières et mémorisées pour définir les contenances – fixes ! - du seau et de la marmite ...

Afin de mettre ce rôle en évidence, on lui associe un nom symbolique, pour assurer une plus grande indépendance de l'algorithme vis-à-vis des valeurs (il ne manipule ainsi que des noms symboliques).

Il va de soi que, comme son nom l'indique, l'association nom-valeur est fixe (est 'constante') et ne peut changer en cours de route (en programmation, une tentative d'affectation d'une valeur à une constante se solde, sinon par une erreur, du moins par un avertissement)

La liste des constantes est énoncée dans la partie déclarative de l'algorithme, selon la syntaxe suivante<sup>3</sup> :

**Constante** *nomDeConstante* = <valeur littérale> [, ...]

*exemple* : Constante pi = 3.14159, tauxTva1 = 0.21,  
tauxTva2 = 0.06, tailleMaxi = 160

Cette place privilégiée tout en tête de l'algorithme permet le cas échéant un accès rapide (et unique) aux valeurs en cas de modification des spécifications du problème (définir un autre terrain pour RobotProg, d'autres contenances de la marmite et du seau pour le Peleur; on peut également imaginer que de temps à autre, des circonstances extérieures entraînent une évolution de certaines valeurs jusque là constantes : p.ex. si la tva passe de 21% à 12%)

<sup>3</sup> les [ ] désignent une partie facultative, et indiquent également une répétition possible; ici donc le mot Constante est associé à une liste dont chaque élément possède trois composantes (un nom, le symbole =, une valeur littérale)



e) EXPRESSIONS

Dès lors que l'on dispose – pour un type donné – d'un ensemble de valeurs littérales, de constantes, de variables et d'opérateurs compatibles, on va pouvoir combiner le tout dans des expressions (comme on le fait en mathématique, sauf qu'en informatique on ne se limitera pas aux seules grandeurs numériques).

Toute expression 'calcule' un résultat (une valeur, ce qui explique qu'on préfère le verbe évaluer à calculer) qui possède un type sans équivoque : ceci impose à toute expression de se trouver dans la partie droite d'une affectation, la partie gauche spécifiant la variable réceptrice du résultat et possédant nécessairement le type approprié.

exemple :

```
Constante prixUnitaire = 1.25, tauxTva : 0.21
Variable quantitéCommandée : Entier, prixHTva, prixTvaC : Réel
...
quantitéCommandée ← 30
prixHTva ← quantitéCommandée * prixUnitaire
prixTvaC ← prixHTva * (1 + tauxTva)
```

*Un œil attentif aura sans doute remarqué dans cet exemple la transgression de la règle du caractère interne des opérateurs : nous avons ici mélangé au moins deux fois les types Entier et Réel dans des expressions. Qu'on se rassure, ils constituent deux types numériques compatibles et la règle est que dans une expression comportant une opérande réelle et une opérande entière, cette dernière (plus 'faible') est convertie en réel (plus 'forte') et que le résultat de l'expression est un réel ; quant à l'affectation finale, nous considérerons ici que c'est le type de la variable réceptrice qui décide : si elle est déclarée réelle, pas de problème ; si elle est déclarée entière, le résultat est tronqué de sa partie décimale avant l'affectation.<sup>6</sup>*

Dans la suite de ce texte, nous utiliserons le mot valeur dans son sens le plus général,

c.-à-d. que si l'on définit une affectation comme **nomDeVariable ← valeur**, alors le mot **valeur** pourra être remplacé par :

- une valeur littérale
- une constante
- une variable
- une fonction (détail sur ce point plus loin)
- toute expression évaluée (qui possède – par 'calcul' - une valeur)

Exemple :

```
Constante CA = 25, CB = 21
Variable a, b : Entier
...
a ← 30                # valeur littérale
b ← CB                # constante
a ← b                 # variable
a ← max(a, b)         # fonction (max détermine et renvoie la plus
                       # grande des deux valeurs transmises)
a ← 25 - CA           # expression (littéral & constante)
b ← a + 8 * CB        # expression (variable, littéral et constante)
```

<sup>6</sup> chaque langage de programmation possède en la matière sa propre règle ; en Pascal, affecter une valeur réelle à une variable déclarée entière entraîne une erreur de compilation

## 2.2. LES TYPES SCALAIRES

On désigne de la sorte (vocabulaire issu des mathématiques) les types pour lesquels constantes et variables ne peuvent contenir qu'une et une seule valeur, et ce, à tout moment !

- ° pour les constantes, cela va de soi, compte tenu de leur définition : une constante conserve sa valeur tout au long de l'algorithme
- ° pour les variables, cela signifie notamment que l'affectation est 'destructrice' (rappelons qu'il s'agit une instruction d'action qui modifie l'état de la variable) :

la valeur contenue par une variable avant affectation est perdue car elle est remplacée par la nouvelle valeur

dans la mesure où une variable ne possède pas de valeur (disons qu'elle est indéfinie) au moment de sa déclaration, dans une séquence algorithmique, on doit nécessairement la trouver d'abord dans une instruction d'affectation où elle figure 'à gauche' avant de pouvoir la trouver ensuite dans une autre instruction où elle figure 'à droite'

```

ne pas faire :   Variable x, y : Entier
                  ...
                  y ← x + 1 # puisque x est indéfini, x + 1 est indéfini !

faire :         Variable x, y : Entier
                  ...
                  x ← 13      # d'abord définir la valeur de x
                  y ← x + 1    # puisque x est défini, x + 1 est défini aussi !
  
```

pour rappel : la partie droite d'une instruction d'affectation est évaluée en premier lieu et son résultat (sa valeur) est affecté ensuite dans la partie gauche (la variable).

```

exemple :      Variable qtéStock, qté : Entier
                  ...
                  qtéStock ← 30          # la variable contient 30
                  qtéStock ← qtéStock * 2 # équivalent à qtéStock ← 30 * 2
                  qté ← qtéStock         # copier (le contenu de) qtéStock dans qté
  
```

```

exemple :      Constante contenanceSeau = 7, contenanceMarmite = 43
                  Variable contenuSeau, contenuMarmite : Entier
                  ...
                  contenuSeau ← 0        # état initial seau
                  contenuMarmite ← 0      # état initial marmite
                  ...
                  contenuSeau ← contenuSeau - 1 # change état seau
                  contenuMarmite ← contenuMarmite + 1 # change état marmite
                  ...
  
```

Note importante : l'affectation n'est pas destructrice pour ce qui se trouve à droite du symbole  $\leftarrow$ ; par exemple l'affectation  $\text{qté} \leftarrow \text{qtéStock}$  place une copie de la variable de droite dans celle de gauche

Nous allons à présent – et rapidement – passer en revue les principaux types scalaires que l'on rencontre usuellement en algorithmique 'classique' : les types Logique, Entier, Caractère, Réel, et Texte (chaînes de caractères)

En programmation (c.-à-d. dans les langages), il y a de très nombreuses variations sur le thème des types scalaires, les langages récents<sup>7</sup> offrent par exemple une 'panoplie' complète de types numériques ...

<sup>7</sup> les langages Python et Ruby proposent à côté des types Entier (avec de multiples sous-types) et Réel (idem) conventionnels, les types Rationnel, Complexe, (domaine de valeurs et opérateurs spécifiques bien sûr) ... etc ...

a) LE TYPE LOGIQUE<sup>8</sup>

À tout seigneur, tout honneur ! on aura compris tout la considération que l'auteur de ce texte porte au type Logique, puisqu'il offre à nos automates la composante « état binaire » qui les rend adaptables, donc programmables ...

Domaine : c'est le type le plus simple de ce point de vue : il se limite à l'ensemble des deux valeurs de vérité {vrai, faux}<sup>9</sup>

Pour les langages de programmation, la question du codage et de la taille de ces deux valeurs ne dépend que de l'implémentation (en théorie, un bit suffirait ; en pratique, on utilise le plus souvent un octet, mais le codage final de vrai et faux est un choix d'implémentation, caché au programmeur)

exemple :

```
Variable estBisseur, trouvé, fini : Logique
```

Littéraux : vrai et faux : **attention** ! ces deux valeurs sont les constantes littérales du domaine Logique; il faut donc considérer vrai et faux strictement comme on le fait avec 25 ou -12 pour le domaine Entier : ce sont des valeurs ! (il ne s'agit donc pas de mots-texte qui devraient alors s'écrire "vrai" et "faux")

exemple :

```
Variable trouvé, fini : Logique      # deux variables logiques (2
                                     états binaires mémorisés)

...
trouvé ← faux                       # affectation de valeur littérale
fini ← vrai                         # affectation de valeur littérale
```

Opérateurs : ce sont ceux de la logique de Boole<sup>10</sup> : NON, ET, OU (ils sont bien internes, puisque toute fonction logique comportant des variables et ces opérateurs logiques retourne vrai ou faux)

La précédence (priorité) de ces opérateurs est la suivante (du plus fort au plus faible) : NON, ET, OU ; si nécessaire, les parenthèses permettent de spécifier un autre ordre d'évaluation

Si v, v1, v2 sont des variables logiques, l'effet des opérateurs NON, ET, OU est décrit par les tables de vérité suivantes :

v	NON v
faux	vrai
vrai	faux

v1	v2	v1 ET v2
faux	faux	faux
faux	vrai	faux
vrai	faux	faux
vrai	vrai	vrai

v1	v2	v1 OU v2
faux	faux	faux
faux	vrai	vrai
vrai	faux	vrai
vrai	vrai	vrai

Parmi les propriétés les plus importantes de ces opérateurs, on retiendra plus particulièrement la loi de De Morgan : (v1 et v2 sont des expressions logiques)

$$\text{NON (v1 OU v2)} \equiv \text{NON v1 ET NON v2}$$

$$\text{NON (v1 ET v2)} \equiv \text{NON v1 OU NON v2}$$

Coupler plusieurs variables (ou expressions) logiques par de tels opérateurs correspond à des situations "multi-états" (pour rappel, un automate dispose habituellement de plusieurs états différents, mais qu'il faut parfois évaluer "ensemble" )

<sup>8</sup> relire le fascicule Éléments de Logique du cours de Mathématiques, n'est pas inutile à ce stade ...

<sup>9</sup> ... et ici, pas question d'utiliser 1 et 0 ! en Pascal (comme dans la plupart des langages de 'haut niveau') récents, ces valeurs sont true et false; en langage C, plus proche de la machine physique, les valeurs vrai/faux sont simulées par les valeurs entières 1/0

<sup>10</sup> dans les langages qui proposent ce type (comme Pascal, mais pas C), le type porte d'ailleurs le nom de booléen

De telles situations ont été rencontrées avec RobotProg :

- être dans un coin particulier s'exprimait p.ex. par : murEnFace ET murAGauche
- ne pas être dans ce coin s'exprime par : NON (murEnFace ET murAGauche),  
ce qui équivaut donc à : (NON murEnFace OU NON murAGauche)
- et en toute généralité, être dans n'importe quel coin a pour expression logique :  
(murEnFace ET (murAGauche OU murADroite)) ce qui s'écrit donc aussi :

NB : dans une boucle (TantQue condition Faire... p.ex.), dans la mesure où la condition correspond à un but à atteindre, donc à un état binaire à rendre vrai, il est habituellement plus clair et lisible de ne pas appliquer de Morgan en programmation (sauf évidemment si cela permet de simplifier l'expression)

donc, si le but est d'aller dans un coin, c.-à-d. de rendre vraie l'expression multi-états : murEnFace ET (murAGauche OU murADroite), on préférera écrire habituellement

TantQue NON (murEnFace ET (murAGauche OU murADroite)) Faire ...

plutôt que (simple application de De Morgan)

TantQue NON murEnFace OU NON (murAGauche OU murADroite) Faire ...

ou que encore (double application de De Morgan)

TantQue NON murEnFace OU (NON murAGauche ET NON murADroite) Faire ...

Note : compte tenu des tables de vérité de OU et de ET, on peut considérer que l'évaluation d'une expression logique s'arrête dès qu'une opérande est vraie dans le cas du OU, et dès qu'une opérande est fausse dans le cas du ET ; c'est ce qui implémenté aujourd'hui dans les 'bons' compilateurs et interpréteurs (cfr. référence du langage)<sup>11</sup>

Affectation : elle devrait être la plus 'directe' possible ; cependant, on voit trop souvent – même chez des programmeurs confirmés - une incompréhension de la nature exacte d'une variable logique, comme ci-dessous :

```
Variable fini : Logique
      x, y : Entier
...
      # affectation préalable de valeurs à x et y
...
      Si x = y Alors      # comparaison des deux variables entières, p.ex.
          fini ← vrai      # affectation valeur littérale
      Sinon
          fini ← faux      # affectation valeur littérale
      finSi
```

5 lignes pour déterminer ce qu'il faut mémoriser dans la variable fini, c'est inutilement lourd ...

retour aux sources :

- qu'est-ce qu'une comparaison ?
- c'est l'utilisation des opérateurs relationnels =, ≠, >, ≥, <, ≤ entre valeurs
- le résultat d'une telle comparaison est toujours vrai ou faux, donc de type Logique

pour cette raison et pour une meilleure clarté de l'algorithme, on préférera écrire directement, (en une ligne !)

fini ← (x = y) # affectation directe du résultat de la comparaison

<sup>11</sup> [http://en.wikipedia.org/wiki/Short-circuit\\_evaluation](http://en.wikipedia.org/wiki/Short-circuit_evaluation)

b) LE TYPE ENTIER

Domaine : l'ensemble des nombres entiers relatifs  $\mathbb{Z}$  (négatifs, nul, positifs)

Pour l'algorithmique : point final !

Pour les langages de programmation, par contre, la double question du codage des valeurs et de la taille de la représentation est ici dépendante de l'implémentation : il est quasi standard de coder les valeurs négatives par la technique du complément vrai<sup>12</sup> ; pour ce qui est de la taille (en octets) des variables – qui, faut-il le rappeler, détermine les bornes du domaine des valeurs représentables<sup>13</sup> – il est impératif de consulter la documentation de référence livrée avec le compilateur ou l'interpréteur du langage ...

*exemple : les différents (sous-)types entiers du compilateur GNU Pascal :*

Type	Range	Size in bytes
Byte	0 .. 255	1
Shortint	-128 .. 127	1
Smallint	-32768 .. 32767	2
Word	0 .. 65535	2
Integer	either smallint or longint	2 or 4
Cardinal	longword	4
Longint	-2147483648 .. 2147483647	4
Longword	0..4294967295	4
Int64	-9223372036854775808 .. 9223372036854775807	8
QWord	0 .. 18446744073709551615	8

Littéraux : les nombres entiers tels qu'ils s'écrivent en arithmétique (254, -33, 0)

Opérateurs : ce sont les opérateurs de l'arithmétique entière qui conservent leur précedence (priorité) naturelle, ainsi que leurs symboles

ces opérateurs doivent être internes (le résultat de toute expression entière est un entier, d'où les deux opérateurs de division div et mod et non pas l'opérateur division 'habituel' / !)

- changement de signe (-)
- addition (+)
- soustraction (-)
- multiplication (\*)
- !! quotient entier (div)      *# 10 div 3 = 3    (10 divisé par 3 donne 3 ...)*
- !! reste de la division entière (mod)      *# 10 mod 3 = 1    (... reste 1)*

NB : l'opérateur modulo (mod) est largement sous-utilisé (ou mal perçu ?) : il a pourtant des caractéristiques particulièrement intéressantes :

- critère de divisibilité :  $n_1$  est divisible par  $n_2$  si  $n_1 \bmod n_2 = 0$
- cas particulier du point précédent : parité :  $n_1$  est pair si  $n_1 \bmod 2 = 0$
- voir plus loin l'exemple 'conversion de secondes' pour d'autres utilisations

À ces opérateurs s'ajoutent quelques fonctions indispensables, notamment valeurAbsolue() qui retourne la valeur absolue de son argument (cfr. la documentation des langages)

```
Variable n1, n2, n3 : Entier,
          pair : Logique
...
n1 ← valeurAbsolue(-13)
n2 ← (n1 * 7) div 5
pair ← (n2 mod 2) = 0
```

<sup>12</sup> cette fois, c'est le fascicule Numération du cours de Mathématique qu'il faut consulter

<sup>13</sup> idem



Propriétés :

- 1) les nombres entiers sont énumérables<sup>14</sup> (même s'il faut un temps infini pour les dénombrer tous !) : en effet ils reposent sur un principe simple : chaque entier peut être obtenu en additionnant/soustrayant 1 à un autre entier (0 par exemple, qui semble un 'bon' point de départ)

le type Entier dispose donc de deux fonctions internes : prédécesseur() et successeur()

```
Variable a, b : Entier
...
a ← 5                # affectation valeur littérale entière
b ← prédécesseur(a)  # affecte 4 à b
```

- 2) compte tenu de leur définition, les entiers sont également comparables entre eux puisqu'ils sont ordonnés (cela découle de leur définition : si  $5 = 4 + 1$ , alors  $5 > 4$  et  $4 < 5$ ),

ces comparaisons s'effectuent avec les opérateurs relationnels  $=, \neq, >, \geq, <, \leq$ , ce qui permettra de les utiliser dans des conditions (et d'en déduire des états binaires).

Une fois encore, nous insistons que tout ce qui précède concerne des valeurs entières au sens large et que donc – pour autant qu'elles soient implicitement ou explicitement entières – on entend par valeurs entières : les littéraux, les constantes symboliques, les variables, les fonctions et les expressions évaluées, on peut donc écrire

```
Constante ca = 33
Variable a, b : Entier
...
a ← 5 + ca           # expression : littéral & constante
b = prédécesseur(2*a + 4) # expression : littéraux & variable
Si a + 1 ≠ b - 2 Alors # comparer des expressions entières
...
FinSi
```

Sous-types : on néglige beaucoup trop souvent cet aspect des choses, en particulier dès la déclaration des variables. Or cette déclaration a un double rôle à jouer :

- c'est d'abord nécessairement une déclaration d'appartenance générique à un type (*telle variable ne peut contenir que des valeurs entières*)
- mais ce devrait être également une déclaration du (sous-)domaine (de valeurs) spécifique au problème posé (*telle variable, parmi les valeurs entières proposées par le type générique, ne peut en contenir que certaines*); cette démarche est déjà une 'annonce' des validations qu'il faudra effectuer plus tard et participe à la documentation de l'algorithme

Grâce à son caractère énuméré, le type Entier permet notamment d'utiliser un sous-type sous forme d'intervalle (précisant la borne inférieure et la borne supérieure) correspondant à *'toute valeur entière comprise entre ... et ...'*; on retrouvera d'autres critères d'utilisations dans le paragraphe consacré aux conditions

*Exemple* : à cette déclaration (trop) générique :

```
Variable jour, mois, heure, minutes, secondes : Entier
```

on préférera (avec un commentaire lorsqu'une situation particulière se présente)

```
Variable  heure : 0..23,
          minutes, secondes : 0..59,
          mois : 1..12,
          jour : 1..31    # !!! borne supérieure 31, 30, 29, 28
```

<sup>14</sup> techniquement, on dira que le type Entier (tout comme les types Logique et Caractère) est un type ordinal, car chaque valeur possède une place précise et immuable au sein du domaine des valeurs ; ce n'est pas le cas du type Réel qui n'est pas ordinal

Lorsque le problème fait apparaître des intervalles spécifiques, mais sujets à possible modification (pas comme ci-dessus où il semble figé que les minutes et secondes restent entre 0 et 59), il est préférable de coupler la possibilité de définir des intervalles à l'utilisation des constantes, cela rend les choses encore plus lisibles et plus facilement adaptables le jour venu

*Exemple* : à cette déclaration

```
Variable age : 18..65
on préférera
Constante ageInférieur = 18, ageSupérieur : 65
Variable age : ageInférieur.. ageSupérieur
```

NB : cette possibilité justifie l'obligation que le bloc déclaratif des constantes précède celui des variables ...

On retrouvera souvent cette notion d'intervalle, car elle apparaîtra sous forme de 'sucre syntaxique' (cfr. plus loin) dans les conditions ...

```
Variable age : 18..65
...
```

ainsi, au lieu d'écrire

```
Si age ≥ 18 et age ≤ 65 Alors ...
```

on aura la possibilité d'écrire (toujours cette recherche de la lisibilité immédiate !)

```
Si age entre 18 et 65 Alors15 ...
```

Exemple : conversion de secondes

on donne un nombre entier de secondes; le convertir en jours, heures, minutes et secondes;  
interface souhaitée : quelque chose comme :

introduisez le nombre de secondes : **xxxxxxx**  
cela correspond à **xx** jours, **xx** heures, **xx** minutes et **xx** secondes

Algorithme convertisseurSecondes :

*# de l'usage de div/mod ...*

```
Variable # à encoder
secondes : Entier
# à calculer
nombreJours, nombreHeures, nombreMinutes, nombreSecondes : Entier
```

Début

*# séquence (future procédure) d'encodage*

```
écrire("introduisez le nombre de secondes : ") # message : invite
lire(secondes) # saisie
```

*# séquence (future procédure) de calcul !! ordre des actions !!*

```
nombreSecondes ← secondes mod 60 # ce qui reste finalement
nombreMinutes ← secondes div 60 # première approche minutes
nombreHeures ← nombreMinutes div 60 # première approche heures
nombreMinutes ← nombreMinutes mod 60 # minutes : ce qui reste finalement
nombreJours ← nombreHeures div 24 # jours : ok
nombreHeures ← nombreHeures mod 24 # heures : ce qui reste finalement
```

<sup>15</sup> sucre particulièrement bienvenu en Pascal où il faut des parenthèses dans les expressions logiques :  
if (age >= 18) and (age <=65) then ... qui peut être remplacé par if age in [18..65] then ...

```
# séquence (future procédure) d'affichage des résultats  
écrire("cela correspond à ", nombreJours, " jours, ",  
      nombreHeures, " heures, ",  
      nombreMinutes, " minutes, ",  
      nombreSecondes)  
  
Fin.
```

### Exercice(s)

à votre avis, dans l'algorithme ci-dessus, pourquoi n'a-t-on pas déclaré directement ceci ?

```
Variable nombreJours : 1..31,  
      nombreHeures : 1..23,  
      nombreMinutes, nombreSecondes : 1..59
```

## c) LE TYPE CARACTÈRE

**Domaine** : pour l'algorithmique, c'est l'ensemble des caractères nécessaires à la communication 'humaine' (lettres majuscules et minuscules, caractères de ponctuation, chiffres, symboles des opérateurs, etc. ...)

en informatique, par contre, (et dans les langages de programmation en particulier), des caractères supplémentaires sont nécessaires pour la communication homme-machine et machine-machine (caractères de contrôle et/ou de commande, p.ex. des caractères tels que <return>, <escape>, <tabulation>, etc. ...)

Le codage et la taille<sup>16</sup> de la représentation des caractères a longtemps été 'simple'<sup>17</sup> : une table de caractères qui s'est standardisée et internationalisée (ASCII 7 bits au départ, 128 caractères disponibles, dont les 32 premiers sont des caractères de contrôle et/ou de commande, ci-dessous à gauche<sup>18</sup>, ASCII 8 bits ensuite, donc 256 caractères, pour permettre l'ajout des caractères accentués, graphiques ou symboliques ; ci-dessous à droite, l'extension des 128 caractères supplémentaires)

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`	128	80	Ç	160	A0	á	192	C0	À	224	E0	α						
1	01	Start of heading	33	21	!	65	41	A	97	61	a	129	81	ù	161	A1	í	193	C1	Á	225	E1	β						
2	02	Start of text	34	22	"	66	42	B	98	62	b	130	82	é	162	A2	ó	194	C2	Â	226	E2	Γ						
3	03	End of text	35	23	#	67	43	C	99	63	c	131	83	â	163	A3	ú	195	C3	Ã	227	E3	π						
4	04	End of transmit	36	24	\$	68	44	D	100	64	d	132	84	ä	164	A4	ñ	196	C4	Ä	228	E4	Σ						
5	05	Enquiry	37	25	%	69	45	E	101	65	e	133	85	å	165	A5	Ñ	197	C5	Å	229	E5	σ						
6	06	Acknowledge	38	26	&	70	46	F	102	66	f	134	86	ä	166	A6	ª	198	C6	Æ	230	E6	μ						
7	07	Audible bell	39	27	'	71	47	G	103	67	g	135	87	ç	167	A7	º	199	C7	Ç	231	E7	τ						
8	08	Backspace	40	28	(	72	48	H	104	68	h	136	88	è	168	A8	¿	200	C8	È	232	E8	φ						
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i	137	89	é	169	A9	¸	201	C9	É	233	E9	θ						
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j	138	8A	ê	170	AA	¸	202	CA	Ê	234	EA	Ω						
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k	139	8B	ï	171	AB	¸	203	CB	Ë	235	EB	σ						
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l	140	8C	î	172	AC	¸	204	CC	Ï	236	EC	ω						
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m	141	8D	ï	173	AD	¸	205	CD	¸	237	ED	ω						
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n	142	8E	ÿ	174	AE	«	206	CE	¸	238	EE	¸						
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o	143	8F	ÿ	175	AF	»	207	CF	¸	239	EF	¸						
16	10	Data link escape	48	30	0	80	50	P	112	70	p	144	90	ÿ	176	BO	¸	208	DO	¸	240	FO	¸						
17	11	Device control 1	49	31	1	81	51	Q	113	71	q	145	91	ÿ	177	B1	¸	209	D1	¸	241	F1	¸						
18	12	Device control 2	50	32	2	82	52	R	114	72	r	146	92	ÿ	178	B2	¸	210	D2	¸	242	F2	¸						
19	13	Device control 3	51	33	3	83	53	S	115	73	s	147	93	ó	179	B3		211	D3	¸	243	F3	¸						
20	14	Device control 4	52	34	4	84	54	T	116	74	t	148	94	ô	180	B4		212	D4	¸	244	F4	¸						
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u	149	95	ö	181	B5		213	D5	¸	245	F5	¸						
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v	150	96	ù	182	B6		214	D6	¸	246	F6	¸						
23	17	End trans. block	55	37	7	87	57	W	119	77	w	151	97	û	183	B7		215	D7	¸	247	F7	¸						
24	18	Cancel	56	38	8	88	58	X	120	78	x	152	98	ÿ	184	B8		216	D8	¸	248	F8	¸						
25	19	End of medium	57	39	9	89	59	Y	121	79	y	153	99	ÿ	185	B9		217	D9	¸	249	F9	¸						
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z	154	9A	ÿ	186	BA		218	DA	¸	250	FA	¸						
27	1B	Escape	59	3B	;	91	5B	[	123	7B	[	155	9B	ÿ	187	BB		219	DB	¸	251	FB	¸						
28	1C	File separator	60	3C	<	92	5C	\	124	7C	\	156	9C	ÿ	188	BC		220	DC	¸	252	FC	¸						
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	]	157	9D	ÿ	189	BD		221	DD	¸	253	FD	¸						
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~	158	9E	ÿ	190	BE		222	DE	¸	254	FE	¸						
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	~	159	9F	f	191	BF		223	DF	¸	255	FF	¸						

Les choses se sont ensuite compliquées avec les systèmes d'exploitation à environnement 'graphique', qui ont multiplié l'internationalisation des jeux de caractères, dont l'évolution et la standardisation est aujourd'hui la codification internationale Unicode (on lira par exemple <http://fr.wikipedia.org/wiki/Unicode> d'abord et <http://unicode.org/fr/> ensuite)

**Littéraux** : tout caractère représentable placé entre apostrophes (lesquelles lèvent l'ambiguïté entre les chiffres et les nombres de 0 à 9 : '5' n'est pas 5 et réciproquement)

Variable c1, c2 : Caractère

...

```
c1 ← 'A'           # affectation valeur littérale
c2 ← c1            # copie de variable à variable
```

dans les langages de programmation, pour spécifier le code d'un caractère de contrôle et de commande (ou de tout autre caractère bien entendu), on peut utiliser le code numérique (la position du caractère dans la table) précédé du symbole #, par exemple, #13 spécifie le retour-chariot (return) et #65 est l'équivalent de 'A', et #48 est l'équivalent de '0' (ce qui confirme bien que '0' n'est pas 0 !)

<sup>16</sup> dans ce cours introductif, nous considérerons qu'il faut un byte (8 bits) pour représenter un caractère, ce qui explique qu'il n'y a que 256 caractères représentables ( $2^8=256$ ) dont les codes vont de 0 à 255, cfr. les tableaux ci-dessus

<sup>17</sup> quoique ... il y a eu longtemps une opposition farouche entre l'univers d'IBM (codage EBCDIC) et le reste de la communauté informatique (codage ASCII)

<sup>18</sup> source : <http://www.cdrummond.qc.ca/cegep/informat/Professeurs/Alain/files>

Opérateurs : il n'y a pas d'opérateur interne sur le type Caractère

Par contre, il existe deux fonctions externes (Caractère  $\leftrightarrow$  Entier) qui établissent une correspondance entre les caractères et leur codification numérique<sup>19</sup> :

```
code numérique entier => littéral      caract(65) => 'A'
littéral => code numérique entier      ordinal('A') => 65

Variable c1, c2 : Caractère
...
c1 ← 'A'                                # affectation valeur littérale
c1 ← caract(65)                          # affectation valeur ordinale
c2 ← caract(ordinal(c1) + 1)             # affectation de 'B' à c2
```

Nous y ajouterons deux fonctions internes (Caractère  $\leftrightarrow$  Caractère) bien pratiques sur les lettres ; si c est un caractère, majusc(c) et minusc(c) donnent respectivement la majuscule et la minuscule de ce caractère : majusc('a')  $\Rightarrow$  'A' et minusc('A')  $\Rightarrow$  'a'

(pour tout autre caractère que des lettres, on considère que la fonction renvoie le caractère inchangé)

```
Variable c1, c2 : Caractère
...
c1 ← 'A'                                # affectation valeur littérale
c2 ← minusc(c1)                          # affectation de 'a' à c2
```

Propriétés : tout comme les entiers, les caractères sont énumérables (et en un temps fini cette fois, puisque leur nombre est fini !) : en effet ils reposent sur un principe simple : chaque caractère possède une 'position' (entière) au sein de la table des caractères représentables (c'est en fait ce qui explique l'existence des fonctions caract et ordinal)

Le type Caractère dispose également donc de deux fonctions internes : prédécesseur() et successeur() : prédécesseur('Z') renvoie 'Y' et successeur('A') renvoie 'B'<sup>20</sup>

Compte tenu de ceci, les caractères sont également comparables entre eux, (avec les opérateurs relationnels =,  $\neq$ , >,  $\geq$ , <,  $\leq$ ), ce qui permettra de les utiliser dans des conditions (et d'en déduire des états).

Sous-types : puisque des entiers 'se cachent' sous les caractères, on retrouve l'intéressante possibilité de définir des intervalles dès la déclaration de variables (intervalles dont les bornes inférieure et supérieure peuvent spécifier des valeurs littérales ou des constantes, mais de type Caractère, cela va de soi !)

Exemple : à cette déclaration (trop) générique :

```
Variable lettreMajuscule, lettreMinuscule, chiffre : Caractère
```

on préférera (avec un commentaire lorsqu'une situation particulière se présente)

```
Variable lettreMajuscule : 'A'..'Z'
      lettreMinuscule : 'a'..'z'
      ..... chiffre : '0'..'9'           # !!! ce ne sont pas des entiers !
```

et grâce aux (sous-types) intervalles, on trouvera des expressions conditionnelles (de comparaison) permettant d'écrire

```
Si chiffre >= '0' et chiffre <= '9' Alors ...
```

de manière plus explicite et en tous cas plus compacte

```
Si chiffre entre '0' et '9' Alors ...
```

<sup>19</sup> En Pascal, ce sont les fonctions char() et ord(); en C, le problème ne se pose pas : les caractères sont traitables explicitement comme des entiers

<sup>20</sup> c'est pour cette raison que dans le langage C il existe cette 'horrible' possibilité : ajouter 1 à un caractère renvoie le caractère 'suivant'

d) LE TYPE RÉEL

Domaine : l'ensemble des nombres réels  $\mathbb{R}$  ]  $-\infty$  ..  $+\infty$  [

Pour l'algorithmique : point final !

Pour les langages de programmation, c'est plus compliqué que pour les entiers : un ordinateur ne saura jamais représenter de réel (puisque ces nombres sont caractérisés par une partie décimale comportant une suite infinie de chiffres) : tout au plus peut on représenter des nombres rationnels avec une 'grande' précision ('grande' étant fonction de l'implémentation)

exemple : si on limite le nombre réel  $\pi$  à 4 décimales (3.1416), c'est le nombre rationnel  $3+1416/10000$  qui est en fait représenté

le codage utilise de manière standard (norme IEEE 754 [http://fr.wikipedia.org/wiki/IEEE\\_754](http://fr.wikipedia.org/wiki/IEEE_754)) la méthode 'mantisse et exposant'

la taille (nombre d'octets) utilisée pour la représentation détermine le nombre de chiffres significatifs<sup>21</sup> (de 7 à 20 selon la taille) ;

pour rappel, le nombre chiffres significatifs reste fixe, ce qui détermine la précision possible pour la représentation des nombres décimaux et l'exactitude des résultats des expressions numériques ;

exemple : si ce nombre est 7, on peut donc représenter (liste non exhaustive)

1234567.0	(sans partie décimale),
123.4567	(partie entière et décimale)
0.1234567	(sans partie entière)

exemple : les différents (sous-)types réels du compilateur GNU Pascal :

Type	Range	Significant digits	Size
Real	platform dependant	???	4 or 8
Single	1.5E-45 .. 3.4E38	7-8	4
Double	5.0E-324 .. 1.7E308	15-16	8
Extended	1.9E-4932 .. 1.1E4932	19-20	10
Comp	-2E64+1 .. 2E63-1	19-20	8
Currency	-922337203685477.5808 .. 922337203685477.5807		8

Littéraux : les nombres 'réels' tels qu'ils s'écrivent en algèbre, soit en notation décimale 2.365, -56.2 (attention, ne pas oublier la présence du point décimal, pour qu'il n'y ait aucune équivoque avec les entiers : 2.0 et non 2) ou en notation scientifique 0.314159E+01

Opérateurs : ce sont les opérateurs de l'algèbre : changement de signe (-), addition (+), soustraction(-), multiplication (\*), division (/), exponentiation (^) qui conservent leur précedence

à ces opérateurs s'ajoutent de très nombreuses fonctions admettant comme arguments des réels (et parfois des entiers) et retournant une valeur réelle (ou entière) ; habituellement, compte tenu de leur nombre et de leur diversité, ces fonctions ne font pas partie du 'noyau' du langage lui-même, mais sont organisées en bibliothèques ('units' en Pascal, 'libs' en C, ...)

- algébriques : valeurAbsolue(), troncature(), arrondi(), ...
- trigonométriques : sin(), cos(), tan() ...
- exponentielles et logarithmiques exp(), log(), ln(), ...
- etc ...

<sup>21</sup> voir vos cours 'scientifiques' et également cet excellent document :

[http://www2.ulg.ac.be/sciences/pedagogique/dossierpds2004/dossier\\_signif.pdf](http://www2.ulg.ac.be/sciences/pedagogique/dossierpds2004/dossier_signif.pdf)

Propriétés : les réels ne sont pas énumérables (le type Réel n'est pas ordinal), il n'y a donc pas de fonctions telles que prédécesseur() ou successeur() ...

par contre, au niveau algorithmique pseudo-code, rien<sup>22</sup> ne nous empêche d'utiliser des intervalles, comme pour les entiers, pour une meilleure lisibilité ...

```
Variable taille : 1.20..2.30 # en mètres, avec 2 décimales exactes
...
```

ainsi, au lieu d'écrire

```
Si taille ≥ 1.20 et taille ≤ 2.30 Alors ...
```

on se donnera la possibilité d'écrire (toujours cette recherche de la lisibilité immédiate !)

```
Si taille entre 1.20 et 2.30 Alors ...
```

important : en toute rigueur, des données de type Réel ne sont pas comparables entre elles !

un petit exemple pour poser le problème :

si l'on dispose de **7** chiffres significatifs, on peut représenter 123.4566, 123.4567, 123.4568, mais pas 123.45672 pas plus que 123.456735 (les décimales supplémentaires sont perdues et ces deux nombres sont mémorisés comme 123.4567 !)

dès lors les comparaisons suivantes sont très dangereuses

```
Variable r1, r2 : Réel
...
r1 ← 123.4567
r2 ← 123.45675
Si r1 ≠ r2 Alors ... # faux !
Si r1 = r2 Alors ... # vrai !
```

en pratique : dans la mesure où ne rédigerons pas des programmes de calcul scientifique et que nous utiliserons des langages (Pascal, C) offrant - à travers leur(s) type(s) réels - un nombre suffisant de chiffres significatifs (au moins 8), nous pourrions considérer (prudemment) que des données réelles sont comparables

<sup>22</sup> côté langages, par contre, aucune expression conditionnelle faisant appel au caractère ordinal n'est acceptée sur des réels

e) LE TYPE TEXTE

Domaine : l'ensemble des chaînes de caractères (textes) que l'on peut constituer à l'aide des caractères du type Caractère

on n'entamera pas ici le débat : le type Texte n'est pas un type scalaire ! dans la mesure où l'on ne cherche pas à atteindre chaque caractère constitutif d'un texte par sa position dans celui-ci (c.-à-d. en refusant de considérer qu'une chaîne est un 'tableau de Caractère'<sup>23</sup>), l'assimilation du type Texte aux types scalaires est défendable, pour autant que les opérateurs, fonctions et autres propriétés soient cohérents de ce point de vue

Littéraux : tout texte formé de caractères et délimité par des guillemets " (si le texte doit lui-même comporter un ", celui-ci doit être doublé)

*exemple* : "ceci est une chaîne de caractères"

"citation : "le corps du délit" fin de citation"

*donc* : Constante erreurType = "erreur de type !",  
message = "programme terminé"

Affectation : pour autant qu'une variable ait été déclarée de type Texte, l'affectation interne et l'affectation externe est légitime

```
Variable chaineEntrée, chaineSortie : Texte
...
lire(chaineEntrée)                # affectation externe
chaineSortie ← "Vous avez introduit : " # affectation interne
écrire(chaineSortie, chaineEntrée)
...
```

Opérateurs : pour respecter notre contrat de type scalaire, nous n'accepterons qu'un seul opérateur : la concaténation (symbole : **&**) qui construit une chaîne de caractères en mettant 'bout à bout' deux autres chaînes de caractères (littérales, constantes, variables, ...)

```
Variable chaine : Texte
...
chaine ← ""                # initialisation d'une chaîne vide
chaine ← "bon" & "jour"    # chaîne contient "bonjour"
chaine ← chaine & " Madame" # chaîne contient "bonjour Madame"
```

À cet opérateur s'ajoutent diverses fonctions admettant comme arguments des chaînes et retournant soit une valeur entière, soit une valeur logique, soit une chaîne selon les cas (les langages regorgent de librairies spécifiques)

ainsi, si ch et sch sont des variables de type Texte

```
Variable ch, sch : Texte
```

- retourne un Entier :

. longueur(ch) : renvoie la longueur de ch en nombre de caractères, 0 si la chaîne est vide

- retourne un Logique (si ch et sch sont bien de type Texte)

. dansChaine(ch, sch) : renvoie vrai si la chaîne sch figure dans la chaîne ch

- retournent une chaîne (si ch est bien de type Texte)

. majuscule(ch), minuscule(ch)

<sup>23</sup> voir un chapitre (très) ultérieur consacré aux structures de tableaux pour plus de détails ...



Propriétés : puisqu'un texte est constitué de caractères et que ceux-ci sont comparables, des données de type Texte sont comparables entre elles (cela correspond à l'ordre lexicographique, extension de l'ordre alphabétique) :

- elles sont égales si elles ont la même longueur et que chacun des caractères lus de gauche à droite sont égaux entre eux ( "coucou" = "coucou" )
- elles sont différentes dès qu'il n'y a plus de correspondance entre deux caractères de même position ( "couucou" ≠ "coucon" puisque 'u' ≠ 'c' )
- une chaîne est 'inférieure' (= avant dans l'ordre lexicographique) ou 'supérieure' (= après dans l'ordre lexicographique) à une autre, si elles sont différentes; c'est le code numérique du caractère qui détermine le caractère < ou > ( "couucou" > "coucon" puisque 'u' > 'c' )
- nb : comme le caractère 'blanc' (l'espace) possède son propre code (32), des chaînes de même contenu mais de longueurs différentes sont différentes, le caractère inférieur ou supérieur dépendant du sens de la comparaison ( "coucou" < "coucou " )
- attention, la comparaison de chaînes de type Texte utilise la codification de manière stricte (p.ex. tient compte de la différence entre majuscules et minuscules : 'A' < 'a' car 65 < 97, et de la différence entre caractères accentués : 'e' < 'é' < 'è' ...)
- pour comparer des chaînes en tenant compte des équivalences, il faudra généralement faire appel à des fonctions spécialisées disponibles dans les bibliothèques associées aux différents langages de programmation

f) CONCRÈTEMENT ...

En programmation, la 'forme concrète' de l'algorithmique sur un ordinateur, le typage des données est un concept essentiel (et incontournable !) : c'est en effet la seule 'clé' qui permet d'interpréter et de traiter correctement une suite de 0 et 1 enregistrée en mémoire centrale

a priori, que faire du contenu de l'octet **01000001** associé à un nom de variable (c.-à-d. stocké à l'adresse-mémoire correspondant à cette variable) ? que vaut-il et que peut-on en faire ?

⇒ s'il est associé à une déclaration de variable telle que

Variable nombre : Entier

c'est un nombre entier codé en binaire dont la valeur est  $65_{10}$ ; dès lors, (le nom de) la variable correspondante sera accepté(e) dans tous les contextes où un nombre est autorisé (expressions, comparaisons, ....)

⇒ s'il est associé à une déclaration de variable telle que

Variable nombre : Caractère

c'est un caractère occupant la  $65^{\text{ème}}$  position de la table de codage, donc un 'A'; la variable correspondante

Autre aspect, comment savoir si les octets consécutifs **01000001 00101100** correspondent à une ou deux variables ?

à nouveau, c'est la déclaration qui est la clé : chaque type de données requiert un certain nombre d'octets pour sa représentation (1, 2, 4, 8, et plus si c'est une donnée de type Texte ...); *un exemple pour le type Entier du Pascal se trouve au § 2.2.b) page 16*

Tous les langages de programmation nécessitent le typage des données, soit directement, soit indirectement ...<sup>24</sup>

⇒ ainsi les langages compilés requièrent-ils – à l'aide d'une syntaxe qui est propre à chacun - une déclaration explicite des constantes et des variables (on parle de typage statique); ci-dessous, quelques déclarations typiques de variables

### Assembleur<sup>25</sup>

il n'y a pas un langage d'assemblage, mais plusieurs : un par type d'architecture (processeur)

un exemple représentatif semble trop compliqué pour être placé ici; à suivre dans le cours d'Organisation et Structure des Données au second semestre ...

### Fortran<sup>26</sup>

les types de données possèdent un identificateur (nom) auquel on associe une taille (en nombre d'octets) ... : identificateur d'abord, noms des variables ensuite ...

INTEGER*2 N1, N2	# deux entiers sur 2 octets
INTEGER*4 N3, N4	# deux entiers sur 4 octets
REAL*4 R1, R2	# deux réels sur 4 octets
REAL*8 R3, R4	# deux réels sur 8 octets
LOGICAL FINI	# un logique (vrai/faux) sur 1 octet
CHARACTER*100 T1	# chaîne de 50 caractères

<sup>24</sup> les commentaires précédés du symbole # ne correspondent pas à la syntaxe native de ces langages ...

<sup>25</sup> pour une initiation 'en douceur' : <http://benoit-m.developpez.com/assembleur/tutoriel/>

<sup>26</sup> pour une initiation 'en douceur' : <ftp://ftp-developpez.com/fortran/cours/hunsinger-cours-complet-fortran.pdf>

Cobol<sup>27</sup>

il n'y a pas d'identificateur de type, mais une description (Picture) des variables en utilisant un préfixe 9 (nombres) ou X (textes) suivi d'une longueur entre parenthèses

```

77 N1  PIC 9(3).           # un entier (base 10) sur 3 chiffres
77 N2  PIC 9(9).           # un entier (base 10) sur 9 chiffres
77 R1  PIC S9(6)V9(3).     # un décimal (base 10) signé avec une partie
                           # entière sur 6 chiffres et avec 3 décimales
77 N3  PIC S9(2) COMP.     # un entier signé (base 2) sur 2 octets
77 T1  PIC X(100).         # un texte sur 100 caractères

```

Pascal

les variables sont associées à un type de données via l'identificateur de celui-ci

```

var n1, n2 : integer;      # deux entiers (base 2) sur 4 octets
var r1, r2 : real;         # deux réels (base 2) sur 4 octets
var fini : boolean;       # un logique
var ch : char;             # un caractère
var txt : string;         # un texte (255 car. maximum)

```

C

les variables sont associées à un type de données via l'identificateur de celui-ci : déclaration 'à la fortran' : identificateur de type d'abord, liste des noms de variables ensuite

```

int n1, n2;                # deux entiers (base 2) sur 4 octets
float r1, r2;              # deux réels (base 2) sur 4 octets
char ch;                   # un caractère
char txt[100]              # un texte sur 100 caractères

```

- ⇒ par contre, les langages interprétés se caractérisent par l'absence de déclaration de variables; c'est lors de l'affectation d'une valeur que le type est déterminé (on parle alors de typage dynamique), via un principe dénommé aujourd'hui 'duck typing' (la valeur détermine le comportement) selon le principe : "when I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck." (*Si je vois un oiseau qui marche comme un canard, nage comme un canard, et cancanne comme un canard, alors j'appelle cet oiseau un canard*)<sup>28</sup>

VB, Python, Ruby ...

```

n1 = 25                    # 25 n'est-il pas un entier ? n1 sera de type entier !
n2 = 25.2                  # et cette valeur est un réel ? n2 sera un réel !
t1 = "hello world"         # ceci ressemble furieusement à un texte ! t1 est ...

```

<sup>27</sup> pour une initiation 'en douceur' : [http://www.podgoretsky.com/ftp/docs/Cobol in 21 Days/index.htm](http://www.podgoretsky.com/ftp/docs/Cobol%20in%2021%20Days/index.htm)

<sup>28</sup> [http://en.wikipedia.org/wiki/Duck\\_typing](http://en.wikipedia.org/wiki/Duck_typing)

## 2.3. QUESTIONS & EXERCICES

- A. Pouvez-vous expliquer ce qu'est un type de données ?
- en 5 phrases ? en 1 phrase ?
  - et sans prendre d'exemple ?
- B. Dans 'variable scalaire', que signifie le mot scalaire ?
- C. Le type de données numérique existe-t-il ? expliquez brièvement
- D. Les expressions numériques sont construites avec des termes et/ou des facteurs : qu'est ce que ces mots signifient ?
- E. Quel est le type et la valeur du résultat (s'il existe, sinon expliquez pourquoi) des expressions suivantes :
- $3 + 8$
  - $3.0 + 8$
  - $3.0 + 8.0$
  - $'3' + '8'$
  - $"3" + "8"$
  - $3 < 8$
  - $'3' < '8'$
  - $"3" < "8"$
  - $(3 < 8)$  ET  $("3" \& "8" = "38")$
- F. Vrai ou faux ? une constante est une valeur littérale qui porte un nom
- G. Vrai ou faux ? il y a une différence entre 'A' et "A" (si vrai laquelle, expliquez ...)
- H. Il y a deux espèces d'affectation (c'est quoi encore l'affectation ?) : lesquelles et pourquoi ? donnez-en des exemples
- I. A votre avis, que se "cache"-t-il en-dessous du concept de type Logique ?

## 2.4. LES TYPES STRUCTURÉS (I)

Opposés aux types scalaires en ceci qu'ils sont composites et peuvent dès lors contenir 'plusieurs' valeurs à tout moment, ils feront l'objet d'une étude approfondie dans un (relativement lointain) chapitre ultérieur (chapitre 5)

### a) LES ENREGISTREMENTS

... le premier 'type' structuré qui sera examiné, ...

### b) LES TABLEAUX

... ensuite, le second ...



### 3. LES INSTRUCTIONS

Si les données (plus précisément les variables) représentent les états d'un automate (l'ensemble de toutes les variables représente l'état global de l'automate, et toute donnée particulière représente un état élémentaire), l'aspect adaptable (programmable) de l'automate demande :

- des instructions d'action (qui modifient ces états : essentiellement, l'affectation) :
  - l'affectation 'interne'
  - l'affectation 'externe'
- des instructions de contrôle (pour modifier le comportement de l'automate en fonction de la valeur des états : les alternatives et les boucles) :
  - la 'conditionnelle'
  - les alternatives 'simple', 'multi-valuée' et 'multi-conditionnée'
  - les boucles à test 'antérieur' et 'postérieur'

#### 3.1. LES INSTRUCTIONS D'ACTION

##### a) L'AFFECTATION

C'est l'action la plus élémentaire (mais la plus fondamentale !) de l'algorithmique impérative : modifier le contenu (actuel) d'une variable en y plaçant une (nouvelle) valeur ; cette instruction se compose de trois parties :

*nomVariable* ← valeur

- un opérateur dont le symbole est ← avec
  - à sa droite une expression (au sens large, comme cela a été précisé plus haut) qui 'calcule' une valeur
  - et à sa gauche le nom de la variable à laquelle cette valeur est affectée.

La variable doit obligatoirement avoir été déclarée au préalable et posséder le type requis pour pouvoir 'recevoir' la valeur de l'expression. Nous nous plaçons résolument ici dans un contexte de 'typage fort' (programmation compilée avec respect strict des types), n'admettant qu'une seule exception : le typage numérique permettant de mélanger entiers et réels dans des expressions (ce 'mélange' renvoyant toujours une valeur de type Réel, ce qui impose à la variable réceptrice de posséder le type Réel).

Il est donc ici totalement exclus d'espérer écrire - par exemple -  $c \leftarrow 'A' + 1$  et ainsi obtenir 'B' (comme c'est le cas dans certains langages comme le C) !

*exemples :*

```
Variable n1, n2 : Entier
          r1, r2 : Réel
          t1, t2 : Texte
          b1, b2 : Logique
...
n1 ← 375                                # affectation de valeur entière
n2 ← (n1 div 173) mod 2                  # affectation de division entière !
r1 ← 0.752                              # affectation de valeur réelle
r2 ← n2 / r1                            # conversion d'abord de n2 en Réel
t1 ← "bon"                              # affectation de chaîne littérale
t2 ← t1 & "jour"                         # affectation de concaténation
b1 ← vrai                                # affectation de valeur logique
b2 ← (t1 = t2)                          # affectation de comparaison
```

b) LES ENTRÉES/SORTIES

Nous désignerons l'affectation correspondant à l'opérateur  $\leftarrow$  par 'affectation interne' ; dans la mesure où même en algorithmique, on doit tenir compte de la communication entre l'algorithme et son environnement extérieur (communication homme-machine), nous devons introduire l'affectation externe' qui permet d'affecter à une variable une valeur 'en provenance de l'extérieur' (par défaut, habituellement, depuis le périphérique d'entrée standard, le clavier ...)

Pour ce faire, nous aurons recours à une procédure **lire**(*nomVariable*) ; on supposera ici que la valeur introduite et la variable ont bien des types compatibles

**lire**(*nomVariable*)

Pour être complet, et de manière à assurer la communication vers l'extérieur (communication machine-homme), citons l'existence d'une procédure de sortie (d'affichage sur le périphérique de sortie standard, habituellement un écran ou une imprimante)

**écrire**(*expression(s)*)

nous ne nous étendrons pas sur le sujet (le lecteur est renvoyé à ses cours de langages)

c) LA SÉQUENCE

C'est un ensemble d'instructions qui s'exécutent dans l'ordre strict où elles ont été rédigées : il s'agit en fait de la définition basique d'un algorithme au sens où c'est ce qu'il faut faire et dans quel ordre pour passer d'un état cohérent de l'algorithme à un autre (et en particulier du problème à sa solution ...)

L'algorithme est évidemment une séquence (appelée séquence principale) qui possède une structure précise et nécessite (comme tout le reste) une partie déclarative; en pseudo-code, nous adopterons l'écriture ci-contre :

cinq mots réservés (dont les équivalents Pascal sont respectivement `program`, `const`, `var`, `begin`, `end`) utilisés dans cet ordre strict possèdent des rôles précis

**Algorithme** *nomAlgorithme* :  
**Constante** *liste de déclarations*  
**Variable** *liste de déclarations*  
**Début**  
     *séquence principale*  
**Fin.**

Algorithme : déclaration de l'ensemble via un nom symbolique

Constante : bloc déclaratif des données 'fixes' (constantes)

Variable : bloc déclaratif des données variables

Début / Fin : deux marqueurs entourant la séquence principale des instructions

Exemple :

```
Algorithme testVente :
Constante tauxTva = 0.21
Variable quantitéVendue : Entier
    prixUnitaire, prixHTva, prixTvaC : Réel
# séquence principale
Début
    # saisie
    écrire("prix unitaire du produit (en €) ? ")    # invite
    lire(prixUnitaire)                             # affectation externe
    écrire("quantité vendue ? ")                  # invite
    lire(quantitéVendue)                          # affectation externe
```



```

# calculs
prixHTva ← quantitéVendue * prixUnitaire           # affectation interne
prixTvaC ← prixHTva * (1 + tauxTVA)                 # affectation interne
# affichage
écrire("prix HTVA : ", prixHTva, " €")
écrire("prix TVAC : ", prixTvaC, " €")
Fin.

```

NB : contrairement aux langages de programmation, la rédaction pseudo-code ne s'embarasse pas d'une syntaxe lourde et contraignante; elle se veut la plus proche possible du problème posé et du raisonnement en langage naturel ...

... dès lors, seuls les éléments structurels indispensables seront utilisés (p.ex. les marqueurs Début / Fin n'apparaîtront qu'au niveau de la séquence principale et nulle part ailleurs, les points-virgules et autres séparateurs rangés au rayon des accessoires, etc. ...)

#### d) CONCRÈTEMENT ...

Quelques exemples représentatifs de la manière dont ce petit algorithme de base testVente s'écrit (déclarations de constantes, de variables, écriture de séquence, ...) dans différents langages représentatifs de leur époque ...

##### Fortran

- affectation interne : utilise le symbole =
- affectation externe : utilise l'instruction `read(5, format) variable :`  
     5 = clavier, *format* = référence à une ligne du programme contenant la description (type, etc ...) de la variable à lire
- affichage, même principe que pour la lecture : `write(6, format) variable : 6=écran`

```

PROGRAM TESTVENTE
! *** declaration de constante
DATA TVA/0.21/
! *** declaration de variables
INTEGER QUANT
REAL PRUNIT, PRHTVA, PRTVAC
! *** formatage des entrees/sorties
11 FORMAT('prix unitaire du produit (en euro) ? ')
12 FORMAT('quantite vendue ? ')
13 FORMAT('prix HTVA : ', F6.2, ' euro')
14 FORMAT('prix TVAC : ', F6.2, ' euro')
21 FORMAT(F5.2)
22 FORMAT(I2)
! *** sequence principale ***
! *** saisie
WRITE(6,11)
READ(5,21) PRUNIT
WRITE(6,12)
READ(5,22) QUANT
! **** calculs
PRHTVA = QUANT * PRUNIT
PRTVAC = PRHTVA * (1 + TVA)
! *** affichage
WRITE(6,13) PRHTVA
WRITE(6,14) PRTVAC
END

```

## Cobol

- au départ, ce langage est très 'littéraire' et utilise des 'verbes' plutôt que des opérateurs (cela rappelle l'assembleur) : MOVE, ADD, SUBTRACT, ... COMPUTE ...
- affectation interne : MOVE valeur TO variable, COMPUTE variable = expression,
- affectation externe : ACCEPT variable (variable déclarée précédemment)
- affichage : DISPLAY valeur

```

IDENTIFICATION DIVISION.
PROGRAM-ID.                TESTVENTE.
*** declaration des donnees *****
DATA DIVISION.
WORKING-STORAGE SECTION.
* pseudo-constante
77 TVA                      PIC 9V99 VALUE 0.21.
* variables
77 QUANTITE                 PIC 999.
77 PRIXUNIT                 PIC 999V99.
77 PRIXHTVA                 PIC 9999V99.
77 PRIXTVAC                 PIC 9999V99.
*** sequence principale *****
PROCEDURE DIVISION.
MAIN SECTION.
**** saisie
    DISPLAY "prix unitaire du produit (en euro) ? ".
    ACCEPT PRIXUNIT.
    DISPLAY "quantité vendue ? ".
    ACCEPT QUANTITE.
**** calculs
    COMPUTE PRIXHTVA = PRIXUNIT * QUANTITE.
    COMPUTE PRIXTVAC = PRIXHTVA * (1.0 + TVA).
**** affichage
    DISPLAY "prix HTVA : " PRIXHTVA " euro".
    DISPLAY "prix TVAC : " PRIXTVAC " euro".
**** fin
    EXIT PROGRAM.
    STOP RUN.

```

## Pascal

- affectation interne via un opérateur explicite :=
- affectation externe : instruction readln(variable)

```

program testVente;
{ exemple de simple séquence }
uses crt;                                { librairie }
const tauxTva = 0.21                     { constante }
var quantiteVendue : integer              { variables }
    prixUnitaire, prixHTva, prixTvaC : real;
{ séquence principale }
begin
    { saisie }
    clrscr;
    write('prix unitaire du produit (en €) ? '); { invite }
    readln(prixUnitaire);                       { affectation externe }
    write('quantité vendue ? ');                 { invite }
    readln(quantiteVendue);                     { affectation externe }
    { calculs }
    prixHTva := quantiteVendue * prixUnitaire;   { affectation interne }
    prixTvaC := prixHTva * (1 + tauxTva);        { affectation interne }
    { affichage }
    writeln('prix HTVA : ', prixHTva, '€');
    writeln('prix TVAC : ', prixTvaC, '€');
    { fin }
    readkey
end.

```

C

```

#include <stdio.h>           // librairie
#include <stdlib.h>          // librairie
#define TAUX TVA 0.21       // constante
// séquence principale
int main() {
    // variables
    int quantiteVendue;
    double prixUnitaire, prixHTva, prixTvaC;
    // saisie
    system("cls");
    printf("prix unitaire du produit (en €) ? ");
    scanf("%lf", &prixUnitaire);
    printf("quantité vendue ? ");
    scanf("%d", &quantiteVendue);
    // calculs
    prixHTva = quantiteVendue * prixUnitaire;
    prixTvaC = prixHTva * (1 + TAUX TVA);
    // affichage
    printf("prix HTVA : %f €\n", prixHTva);
    printf("prix TVAC : %f €\n", prixTvaC);
    // fin
    system("pause");
    return 0;
}

```

Ruby

```

#!/usr/bin/env ruby
TVA = 0.21 # constante
# *** séquence principale ***
# saisie
puts "prix unitaire du produit (en €) ? "
prixUnitaire = gets.to_f
puts "quantité vendue ? "
quantite = gets.to_f
# calculs
prixHTva = prixUnitaire * quantite
prixTvaC = prixHTva * (1 + TVA)
# affichage
puts "prix HTVA :#{prixHTva}\n"
puts "prix TVAC :#{prixTvaC}\n"
# *****

```

e) LA PROCÉDURE

Outil fondamental de l'algorithmique (et de la programmation) impérative procédurale, on présente souvent la procédure comme la généralisation du concept d'action.

Elle permet en effet de donner un nom à une séquence d'instructions, de 'sortir' celle-ci de la séquence principale (elle est déclarée et décrite séparément) et de la faire exécuter en invoquant simplement son nom; il y a donc une instruction d'action (implicite) : l'appel de procédure

Exemple : une écriture non procédurale ...

```

Algorithme testVente :
Constante tauxTva = 0.21
Variable quantitéVendue : Entier
      prixUnitaire, prixHTva, prixTvaC : Reel
# séquence principale
Début
  # saisie
  écrire("prix unitaire du produit (en €) ? ")
  lire(prixUnitaire)
  écrire("quantité vendue ? ")
  lire(quantitéVendue)
  # calculs
  prixHTva ← quantitéVendue * prixUnitaire
  prixTvaC ← prixHTva * (1 + tauxTVA)
  # affichage
  écrire("prix HTVA : ", prixHTva, " €")
  écrire("prix TVAC : ", prixTvaC, " €")
Fin.

```

... s'écrira procéduralement comme suit :

```

Algorithme testVente :
Constante tauxTva = 0.21
Variable quantitéVendue : Entier
      prixUnitaire, prixHTva, prixTvaC : Reel
#
Procédure saisirDonnées :
Début
  écrire("prix unitaire du produit (en €) ? ")
  lire(prixUnitaire)
  écrire("quantité vendue ? ")
  lire(quantitéVendue)
Fin;
#
Procédure calculerRésultats :
Début
  prixHTva ← quantitéVendue * prixUnitaire
  prixTvaC ← prixHTva * (1 + tauxTVA)
Fin;
#
Procédure afficherRésultats :
Début
  écrire("prix HTVA : ", prixHTva, " €")
  écrire("prix TVAC : ", prixTvaC, " €")
Fin;
# séquence principale
Début
  saisirDonnées
  calculerRésultats
  afficherRésultats
Fin.

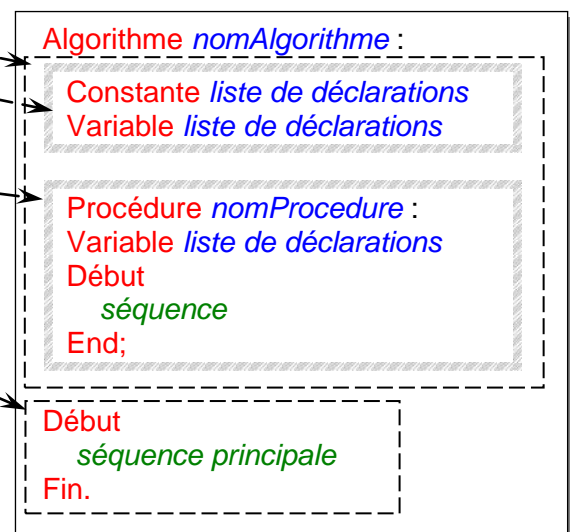
```

Les avantages de l'approche procédurale sont nombreux (on y reviendra plus en détail dans un prochain chapitre consacré aux procédures et aux fonctions)

- ⇒ meilleure structuration de la pensée : la technique de l'analyse descendante (top-down en anglais) consiste à aborder les problèmes à résoudre en les décomposant en sous-problèmes; dans l'exemple ci-dessus, le problème : calculer la facture d'une commande comporte trois parties : la saisie des données, le calcul des montants et l'affichage des résultats
- ⇒ indépendance : chacun des sous-problème peut être traité de manière autonome : la saisie peut ignorer la destination des données; le calcul ne se préoccupe pas de la manière dont les données ont été introduites, ni ce qui sera fait des résultats; l'affichage n'a que faire de l'origine des données à afficher; ce qui importe c'est que les choses soient faites dans l'ordre (c'est le rôle de la séquence principale) et qu'il existe une 'zone commune' que se partageront les procédures : les constantes et variables (le bloc déclaratif des données)<sup>29</sup>
- ⇒ adaptabilité : le jour où des modifications sont nécessaires, il est plus simple de modifier une procédure
- ⇒ réutilisabilité : on en donnera des exemples plus tard<sup>30</sup> : avoir des procédures spécialisées et indépendantes permet une grande souplesse et une modularité accrue; une procédure bien écrite a de grandes chances de pouvoir être invoquée dans d'autres procédures ou à différents endroits de l'algorithme principal
- ⇒ abstraction : on remarque entre les deux versions de la séquence principale du problème une différence notable : dans la seconde, l'algorithme est plus abstrait (il ne fait aucune référence aux données), il se préoccupe plus du "what" que du "how" (cette tâche étant déléguée aux procédures)<sup>31</sup>

Côté pseudo-code, la structure générale d'un algorithme est la suivante : nous adoptons ici un principe que l'on retrouve tel quel en programmation Pascal : tout est toujours déclaré et défini avant son utilisation : on trouve donc, après la déclaration/nommage de l'algorithme lui-même

- ⇒ un bloc ~~déclaratif~~ comportant lui-même et dans cet ordre
  - un bloc ~~déclaratif des données~~
    - déclaration de constantes
    - déclaration des variable
  - un bloc ~~déclaratif de toutes les procédures~~
- ⇒ un bloc 'de code' correspondant à la séquence principale



<sup>29</sup> dans un premier temps en tout cas (variables globales), on reviendra amplement sur cette considération lors de l'étude des paramètres

<sup>30</sup> mais l'expérience robotProg l'a largement illustrée

<sup>31</sup> souvenez-vous – dans la première partie de ce cours- de la manière dont l'algorithme abstrait "peleurDePatates" a pris une forme concrète

f) CONCRÈTEMENT ...

Au cours de l'histoire et l'évolution des langages, la nécessité des procédures s'est traduite à l'aide d'outils possédant des syntaxes très diverses ...

Fortran

- c'est un peu lourd à implémenter<sup>32</sup> : les procédures sont décrites (rédigées) à la suite de la séquence principale; elles sont annoncées par le mot réservé `SUBROUTINE` et invoquées par l'instruction `CALL`, les variables qu'elles manipulent y sont (re)déclarées explicitement
- tant au niveau du programme principal que de chacune des procédures, il faut spécifier – après déclaration des variables – celles qui sont partagées (communes, globales) par le programme et les procédures : c'est le rôle de la ligne `COMMON`

```

PROGRAM FACTURE
C
C *** declaration des variables
C
      INTEGER QUANT
      REAL TVA, PRUNIT, PRHTVA, PRTVAC
C
C *** globalisation des variables (partage avec les procedures)
C
      COMMON QUANT, TVA, PRUNIT, PRHTVA, PRTVAC
C
C *** programme principal
C
      TVA = 0.21
      CALL SAISIR
      CALL CALCULER
      CALL AFFICHER
      END
C
C *** procedures
C
      SUBROUTINE SAISIR
      INTEGER QUANT
      REAL TVA, PRUNIT, PRHTVA, PRTVAC
      COMMON QUANT, TVA, PRUNIT, PRHTVA, PRTVAC
11      FORMAT('prix unitaire du produit (en euro) ? ')
12      FORMAT('quantite vendue ? ')
21      FORMAT(I2)
22      FORMAT(F5.2)
      WRITE(6,11)
      READ(5,21) QUANT
      WRITE(6,12)
      READ(5,22) PRUNIT
      END

      SUBROUTINE CALCULER
      INTEGER QUANT
      REAL TVA, PRUNIT, PRHTVA, PRTVAC
      COMMON QUANT, TVA, PRUNIT, PRHTVA, PRTVAC
      PRHTVA = QUANT * PRUNIT
      PRTVAC = PRHTVA * (1 + TVA)
      END

      SUBROUTINE AFFICHER
      INTEGER QUANT
      REAL TVA, PRUNIT, PRHTVA, PRTVAC
      COMMON QUANT, TVA, PRUNIT, PRHTVA, PRTVAC
13      FORMAT('prix HTVA : ', F6.2, ' euro')
14      FORMAT('prix TVAC : ', F6.2, ' euro')
      WRITE(6,13) PRHTVA
      WRITE(6,14) PRTVAC
      END

```

<sup>32</sup> dans un chapitre ultérieur, de meilleurs techniques seront illustrées (procédures, fonctions et paramètres)

## Cobol

- ce langage est par nature procédural (c'est sans doute le premier qui implémente le concept de "programmation structurée top-down") : les procédures sont désignées par le terme de "paragraphes" et sont invoquées au moyen de l'instruction PERFORM
- toutes les variables étant globales, la structuration est donc très simple, les données déclarées en tête de programme étant visibles partout de l'intérieur
- il n'y a pas de concept de 'variable locale' à une procédure

```

IDENTIFICATION DIVISION.
PROGRAM-ID.                                FACTURE.
*
*** declaration des donnees *****
*
DATA DIVISION.
WORKING-STORAGE SECTION.
*pseudo-constante
77 TVA                                PIC 9V99 VALUE 0.21.
* variables
77 QUANTITE                            PIC 999.
77 PRIXUNIT                            PIC 999V99.
77 PRIXHTVA                            PIC 9999V99.
77 PRIXTVAC                            PIC 9999V99.
*
*** sequence principale *****
*
PROCEDURE DIVISION.
MAIN SECTION.
    PERFORM SAISIR.
    PERFORM CALCULER.
    PERFORM AFFICHER.
    EXIT PROGRAM.
    STOP RUN.
*
*** procedures
*
SAISIR.
    DISPLAY "prix unitaire du produit (en euro) ? ".
    ACCEPT PRIXUNIT.
    DISPLAY "quantité vendue ? ".
    ACCEPT QUANTITE.

CALCULER.
    COMPUTE PRIXHTVA = PRIXUNIT * QUANTITE.
    COMPUTE PRIXTVAC = PRIXHTVA * (1.0 + TVA).

AFFICHER.
    DISPLAY "prix HTVA : " PRIXHTVA " euro".
    DISPLAY "prix TVAC : " PRIXTVAC " euro".
*
*****

```

## Pascal

- ce langage est par nature procédural : les procédures<sup>33</sup> sont rédigées séparément mais avant la séquence principale (dans un bloc déclaratif/descriptif prenant place entre le bloc déclaratif des données et la séquence principale) : elles sont ensuite invoquées en spécifiant tout simplement leur nom
- toutes les variables déclarées en tête de programme sont considérées comme globales et donc accessibles de l'intérieur des procédures
- si nécessaire, des variables peuvent être déclarées dans une procédure (non illustré ci-dessous); elles sont alors locales à cette procédure (et inaccessibles en dehors de cette procédure)

```

program facture;
uses crt;
const tva = 0.21;

{ *** variables globales ***** }

var quantite : integer;
    prix_unitaire, total_ht, total_ttc : real;

{ *** procedures ***** }

procedure saisir;
begin
    write('prix unitaire de l'article (en euro) ? ');
    readln(prix_unitaire);
    write('combien d'articles voulez-vous ? ');
    readln(quantite);
end;

procedure calculer;
begin
    total_ht := prix_unitaire * quantite;
    total_ttc := total_ht * (1 + tva);
end;

procedure afficher;
begin
    writeln;
    writeln('total ht : ',total_ht:6:2, ' euro');
    writeln('total ttc : ',total_ttc:6:2, ' euro');
end;

procedure terminer;
begin
    writeln;
    write('appuyez sur une touche pour continuer ... ');
    readkey;
end;

{ *** sequence principale ***** }

begin
    saisir;
    calculer;
    afficher;
    terminer;
end.

```

<sup>33</sup> il y a deux outils distincts de structuration (aux rôles bien définis) : la procédure et la fonction (on y reviendra dans le présent cours et dans le cours consacré à ce langage)



C

- ce langage est par nature procédural : les fonctions<sup>34</sup> sont rédigées séparément et invoquées par leur nom (comme en Pascal)
- dans le respect de la norme ANSI, les fonctions sont d'abord 'annoncées' avant la séquence principale (déclaration de prototype spécifiant leur nom); étant connues, elles peuvent ainsi être invoquées dans la séquence principale; leur 'corps' (c.-à-d. la rédaction de leur code) n'intervient qu'après la séquence principale
- contrairement à ce qui est fait ci-dessous, il n'est pas de bonne pratique d'utiliser des variables globales en C (c.-à-d. des données déclarées en dehors de la séquence principale) : tout se règle par passage de paramètres entre fonctions ...

```
// exemple de 'procedures' en C

#include <stdio.h>          // librairie
#include <stdlib.h>         // librairie

#define TAUX TVA 0.21      // constante

// variables globales

int quantiteVendue;
double prixUnitaire, prixHTva, prixTvaC;

// prototypes des fonctions-procedures

void saisir();
void calculer();
void afficher();

// séquence principale

int main() {
    saisir();
    calculer();
    afficher();

    system("pause");
    return 0;
}

// definition des fonctions-procedures

void saisir() {
    system("cls");
    printf("prix unitaire du produit (en €) ? ");
    scanf("%lf", &prixUnitaire);
    printf("quantité vendue ? ");
    scanf("%d", &quantiteVendue);
}

void calculer() {
    prixHTva = quantiteVendue * prixUnitaire;
    prixTvaC = prixHTva * (1 + TAUX TVA);
}

void afficher() {
    printf("prix HTVA : %f €\n", prixHTva);
    printf("prix TVAC : %f €\n", prixTvaC);
}
```

<sup>34</sup> il n'y a qu'un seul outil 'technique' : la fonction C (pouvant jouer un rôle de procédure stricte, de fonction stricte, ou d'un mix des deux) (on y reviendra dans le cours consacré à ce langage)

### 3.2. LES INSTRUCTIONS DE CONTRÔLE

Si les instructions d'action (affectation) sont utilisées pour modifier les états (variables) de l'algorithme, il faut des instructions de contrôle pour pouvoir modifier (contrôler) le comportement de l'algorithme sur base de ses états

#### a) LES CONDITIONS

En toute généralité, une condition est une expression dont la valeur est de type Logique

Peuvent donc convenir comme conditions : *(nous utiliserons les déclarations suivantes pour illustrer notre propos avec un simple Si ... Alors ...)* :

```
Constante limiteInférieure = 120, limiteSupérieure = 1000
Variable limite, contenu : Entier
        limiteOK : Logique
        lettre : Caractère
```

- des comparaisons directes entre valeurs (littérales, constantes ou variables des types scalaires décrits précédemment) sur base des opérateurs relationnels **=, ≠, >, ≥, <, ≤**

```
Si limite = 234 Alors ...
Si limite < limiteSupérieure Alors ...
Si contenu <= limite Alors ...
Si lettre ≠ 'N' Alors ...
```

- des variables logiques, déclarées comme telles, auxquelles on aura affecté directement ou indirectement (p.ex. par comparaison directe) la valeur vrai ou faux

```
limiteOK ← vrai # affectation directe
Si limiteOK Alors ...
limiteOK ← (limite >= limiteInférieure
            ET limite <= limiteSupérieure) # affectation indirecte
Si limiteOK Alors ...
```

- des expressions logiques utilisant les opérateurs **NON, ET, OU** et dont les opérandes sont de type logique

```
Si limiteOK ET (lettre = 'M' OU lettre = 'm') Alors ...
Si NON (limiteOK OU contenu < 20) Alors ...
```

- des fonctions dont la valeur de retour est de type logique *(on découvrira l'outil fonction plus tard dans ce texte ...)*

```
Si dansChaine("algorithmique", "algo") Alors ...
```

- l'appartenance d'une variable à un intervalle<sup>35</sup> (uniquement pour des données de type énuméré : Entier et Caractère) à l'aide des clauses **ENTRE** et **ET** qui spécifient les bornes inférieure et supérieure de l'intervalle (comprises), qui peuvent être des valeurs littérales, des constantes, des variables, etc. ...) : *si la valeur de la variable est entre ... et ...*

```
Si lettre ENTRE 'A' ET 'Z' Alors ...
Si limite ENTRE limiteInférieure ET limiteSupérieure Alors ...
```

cette clause permet simplement une meilleure expressivité (toujours cette recherche du meilleur niveau d'abstraction possible); elle est une forme plus légère de son équivalent de base

```
Si lettre ≥ 'A' ET lettre ≤ 'Z' Alors ...
Si limite ≥ limiteInférieure ET limite ≤ limiteSupérieure Alors ...
```

<sup>35</sup> en Pascal : if lettre in ['A'..'Z'] then ...

- l'appartenance d'une variable à une liste de valeurs<sup>36</sup> (uniquement pour des données de type énuméré : à l'aide de la clause **PARMI** qui permet de spécifier entre parenthèses une liste uniquement constituée de valeurs littérales) : *si la valeur de la variable est parmi ...*

Si lettre **PARMI** ('M', 'm', 'F', 'f') Alors ...

meilleure expressivité ici, où le gain d'écriture par rapport à la forme basique est particulièrement évident ...

Si lettre = 'M' OU lettre = 'm' OU lettre = 'F' OU lettre = 'f' Alors ...

- Remarque : permettre à l'utilisateur d'encoder aussi bien des majuscules que des minuscules entraîne un alourdissement pénible des expressions conditionnelles; pour une simplification bienvenue de celles-ci, on se souviendra des fonctions 'caractère' majusc() et minusc()

Variable sexe : Caractère

...

écrire("introduisez le sexe : (m/f) : ")

lire(sexe)

# on remplacera ...

Si sexe = 'M' OU sexe = 'm' OU sexe = 'F' OU sexe = 'f' Alors ...

# ... par ...

sexe ← minusc(sexe)

Si sexe = 'm' OU sexe = 'f' Alors ...

# ... ou mieux encore par ...

sexe ← minusc(sexe)

Si sexe parmi ('m', 'f') Alors ...

Une condition possède une valeur de type Logique (vrai/faux); bien évidemment, sa négation possède sa valeur inverse (faux/vrai)

C'est sans doute l'occasion de (re)repréciser quelques propriétés de la négation ainsi que quelques éléments de syntaxe pseudo-code, en particulier l'usage des parenthèses associé à la négation

opérateur	négation
=	≠
≠	=
<	≥
≤	>
>	≤
≥	<
(c1 OU c2)	NON (c1 OU c2) ≡ NON c1 ET NON c2
(c1 ET c2)	NON (c1 ET c2) ≡ NON c1 OU NON c2
n ENTRE n1 ET n2	NON (n ENTRE n1 ET n2)
n PARMI (...)	NON (n PARMI (...))

<sup>36</sup> En Pascal : if lettre in ('M', 'm', 'F', 'f') then ...

b) LES STRUCTURES CONDITIONNELLES ET ALTERNATIVES

Les instructions alternatives sont la forme fondamentale des instructions de contrôle ; elles permettent de modifier le comportement de l'algorithme en fonction d'un état logique binaire (c.-à-d. une condition, cfr. précédemment)

➤ Si ... Alors ... [ Sinon ... ]

L'instruction conditionnelle est la forme la plus simple, elle se compose d'une condition (au sens général défini précédemment) et d'une seule séquence (également au sens général)<sup>37</sup> à exécuter *si et seulement si* cette condition est vraie ; elle a pour syntaxe :

**Si** [non] condition **Alors**  
séquence  
**finSi**

Exemple :

```
Variable nombre1, nombre2 : Réel
...
lire(nombre1)
lire(nombre2)
Si nombre2 ≠ 0 Alors
    écrire(nombre1 / nombre2)
finSi
...
```

L'instruction alternative permet de spécifier une séquence à exécuter si la condition est vraie et une autre séquence à exécuter si la condition est fausse ; elle a pour syntaxe :

**Si** [non] condition **Alors**  
séquence1  
**Sinon**  
séquence2  
**finSi**

Exemple :

```
Variable sexe, etatCivil : Caractère
...
lire(sexe)
Si minusc(sexe) PARM ('h', 'f') Alors
    Si minusc(sexe) = 'h' Alors
        écrire("Bonjour Monsieur !")
    Sinon
        lire(etatCivil)
        Si minusc(etatCivil) PARM ('c', 'm') Alors
            Si minusc(etatCivil) = 'c' Alors
                écrire("Bonjour Mademoiselle !")
            Sinon
                écrire("Bonjour Madame !")
            finSi
        finSi
    Sinon
        écrire("erreur d'encodage pour état civil !")
    finSi
finSi
Sinon
    écrire("erreur d'encodage pour sexe !")
finSi
...
```

<sup>37</sup> « sens général » pour les conditions et pour les actions que nous ne rappellerons plus à l'avenir ...

Comme on le remarque, l'imbrication des instructions alternatives est tout à fait possible et permet - sans équivoque grâce à la présence des marqueurs de fin (finSi) – d'exprimer des situations complexes, par exemple des alternatives imbriquées où la partie 'interne' ne possède pas de Sinon ...<sup>38</sup>

Exemple :

```

Si condition1 Alors
    Si condition2 Alors
        séquence1
    FinsSi
Sinon
    séquence2
finSi

```

Une alternative dont la branche Alors est vide n'est pas autorisée; on aura dès lors recours à la négation, afin de retrouver une conditionnelle simple

```

Variable nombre1, nombre2 : Réel
...
lire(nombre1)
lire(nombre2)
Si nombre2 = 0 Alors
    écrire("erreur : diviseur nul")
Sinon
    écrire(nombre1 / nombre2)
finSi
...

```

Si on ne veut pas du message d'erreur,

on ne pourra pas écrire

```

Si nombre2 = 0 Alors
Sinon
    écrire(nombre1 / nombre2)
finSi
...

```

on devra écrire (remarquer le recours à la négation)

```

Si nombre2 ≠ 0 Alors
    écrire(nombre1 / nombre2)
finSi
...

```

<sup>38</sup> à ce niveau, le Pascal est loin d'être idéal, l'imbrication de if ... then ... else peut conduire à de véritables catastrophes logiques qui ne peuvent être évitées qu'en surajoutant des begin ... end à des endroits où ils n'apparaissent pas nécessaires à première vue : pour coder l'exemple du haut de la page, si on écrit :

```

if condition1 then
    if condition2 then
        séquence1
    else séquence2 ;

```

la logique est fautive : ce n'est pas parce qu'on a associé par indentation le else avec le premier if que c'est ce qui se passe en réalité : un else est toujours associé avec le if qui le précède (en rouge ci-dessus), et comme on ne peut pas écrire

```

if condition1 then
    if condition2 then
        séquence1 → (* pas de ; devant un else)
    else séquence2 ;

```

il faut bien se résoudre à écrire ☹ :

```

if condition1 then begin
    if condition2 then
        séquence1 end
    else séquence2 ;

```

Ces situations à conditions multiples (dont souvent des validations de données à assurer en premier lieu) mettent souvent le rédacteur d'un algorithme dans un état de grande perplexité : comment ne pas exécuter la suite si les conditions initiales ne sont pas remplies ?

*Exemple* : demander le sexe, l'âge, la taille et le poids de la personne pour calculer son IMC : indice de masse corporelle, défini comme poids (en kg) / taille<sup>2</sup> (en m) ; ce calcul ne doit pas s'effectuer pour une femme enceinte ni pour des enfants (< 18 ans) ni des personnes âgées (> 65 ans) ; on souhaite que l'interface homme-machine soit le plus réactif possible ('sortir' aussitôt qu'une situation anormale se présente)

```

variable sexe, enceinte : Caractère
variable age, poids : Entier
variable taille, imc : Réel
...
écrire("quel est le sexe de la personne (h/f) ? ")
lire(sexe)
Si minusc(sexe) parmi ('h', 'f') Alors
  Si minusc(sexe) = 'h' Alors
    écrire("quel est l'âge de la personne (18..65) ? ")
    lire(age)
    Si age entre 18 et 65 alors
      lire(taille)
      lire(poids)
      imc ← poids / (taille * 100 * taille * 100)
    Sinon
      écrire("hors étude vu l'age")
    finSi
  Sinon
    écrire("la personne est-elle enceinte (o/n) ? ")
    lire(enceinte)
    Si minusc(enceinte) parmi ('o', 'n') Alors
      Si minusc(enceinte) = 'n' Alors
        écrire("quel est l'âge de la personne (18..65) ? ")
        lire(age)
        Si age entre 18 et 65 alors
          lire(taille)
          lire(poids)
          imc ← poids / (taille * 100 * taille * 100)
        Sinon
          écrire("hors étude vu l'age")
        finSi
      Sinon
        écrire("hors étude car enceinte")
      finSi
    Sinon
      écrire("erreur de valeur enceinte")
    finSi
  finSi
Sinon
  écrire("erreur de valeur sexe")
finSi

```

la présence redondante du bloc encadré est peu acceptable (cela alourdit inutilement l'algorithme) et cacher ce bloc dans une procédure (cela se verrait évidemment moins ☺) l'est tout autant ☹, du moins pour un puriste !

une méthode assez efficace consiste à considérer d'abord les situations les plus restrictives (ici, les femmes à cause de la sous question sur la grossesse) :

```

Variable sexe, enceinte : Caractère
      age, poids : Entier
      taille, imc : Réel
...
lire(sexe); sexe ← minusc(sexe)
Si sexe parmi ('h', 'f') Alors
  Si sexe = 'f' Alors
    lire(enceinte); enceinte ← minusc(enceinte)
    Si enceinte parmi ('o', 'n') Alors
      Si enceinte = 'o' Alors
        écrire("hors étude car enceinte")
      finSi
    Sinon
      écrire("erreur de valeur enceinte")
    finSi
  finSi
  #
  # ici, comme on a validé le sexe, et qu'on a traité le cas de la femme enceinte
  # on est sûr d'avoir un homme ou une femme non enceinte et de pouvoir continuer
  # à condition d'ajouter un test de validation complémentaire
  #
  Si sexe = 'h' ou enceinte = 'n' Alors
    lire(age)
    Si age entre 18 et 65 Alors
      lire(taille)
      lire(poids)
      imc ← poids / (taille * 100 * taille * 100)
    Sinon
      écrire("hors étude vu l'age")
    finSi
  finSi
Sinon
  écrire("erreur de valeur sexe")
finSi

```

dans certains cas, quand la batterie de Si imbriqués ont une même variable en commun (testée pour différentes valeurs), il est plus clair de jouer sur la présentation

```

Variable jour : 1..7
...
lire(jour)
Si jour = 1 Alors
  écrire("lundi")
Sinon
  Si jour = 2 Alors
    écrire("mardi")
  Sinon
    Si jour = 3 Alors
      écrire("mercredi")
    Sinon
      Si jour = 4 Alors
        écrire("jeudi")
      Sinon
        Si jour = 5 Alors
          écrire("vendredi")
        Sinon
          Si jour = 6 Alors
            écrire("samedi")
          Sinon
            écrire("dimanche")
          finSi
        finSi
      finSi
    finSi
  finSi
  # ☺
finSi
  # ☺☺
finSi
  # ☺☺☺
finSi
  # ☺☺☺☺
finSi
  # ☺☺☺☺☺
...

```

```

Variable jour : 1..7
...
lire(jour)
Si jour = 1      Alors écrire("lundi")
  SinonSi jour = 2 Alors écrire("mardi")
  SinonSi jour = 3 Alors écrire("mercredi")
  SinonSi jour = 4 Alors écrire("jeudi")
  SinonSi jour = 5 Alors écrire("vendredi")
  SinonSi jour = 6 Alors écrire("samedi")
  Sinon :        écrire("dimanche")
finSi
...

```

le SinonSi est une écriture qu'on peut utiliser dans de tels cas, elle permet d'éviter d'écrire une suite de finSi au bout de l'instruction et de la remplacer par un seul finSi global

### ➤ Concrètement ...

Au début, les premiers langages étaient l'image du jeu d'instructions du processeur; il faut attendre 1962 et la version Fortran IV pour y disposer de la première forme rudimentaire (ô combien) du IF (CONDITION) associé au GOTO ! : il n'a pas de THEN explicite et encore moins de ELSE !

C'est le début d'une (longue, plus de 15 ans !) période de 'programmation spaghetti'<sup>39</sup>; bien entendu, on peut avoir recours aux procédures (SUBROUTINES), mais comme on l'a entrevu précédemment, l'aspect déclaration de données est particulièrement lourd (redéclarations systématiques, lignes de COMMON, ...)

#### Fortran

```

PROGRAM ALTERNATIVE

c *** declaration des variables
      CHARACTER SEXE, ETAT

c *** formats d'entree/sortie
11      FORMAT('entrez le sexe (M/F) ')
12      FORMAT('etes-vous mariee (O/N) ?')
15      FORMAT('erreur sexe ! stop')
16      FORMAT('erreur etat-civil ! stop')
17      FORMAT('bonjour Monsieur')
18      FORMAT('bonjour Madame')
19      FORMAT('bonjour Mademoiselle')
21      FORMAT(1X)

c *** programme principal
      WRITE(6, 11)
      READ(5, 21) SEXE
      IF ((SEXE.EQ.'M').OR.(SEXE.EQ.'m')) GOTO 100
      IF ((SEXE.EQ.'F').OR.(SEXE.EQ.'f')) GOTO 120
      WRITE(6, 15)
      GOTO 999
100     WRITE(6, 17)
      GOTO 999
120     WRITE(6, 12)
      READ(5, 21) ETAT
      IF ((ETAT.EQ.'O').OR.(ETAT.EQ.'o')) GOTO 200
      IF ((ETAT.EQ.'N').OR.(ETAT.EQ.'n')) GOTO 220
      WRITE(6, 16)
      GOTO 999
200     WRITE(6, 18)
      GOTO 999
220     WRITE(6, 19)
      GOTO 999
999     END

```

Si la condition est vraie, on va à la ligne désignée par le GOTO, sinon on passe à la ligne suivante (l'absence d'un équivalent du Sinon et du FinSi entraîne des GOTO supplémentaires)

Remarquer ☺ les opérateurs relationnels .EQ., .NE., .LT. etc... et les opérateurs logiques .AND., .OR. et .NOT. qui rappellent les instructions du langage machine

<sup>39</sup> [http://fr.wikipedia.org/wiki/Programmation\\_spaghetti](http://fr.wikipedia.org/wiki/Programmation_spaghetti)



## Cobol

Au début des années 1960, alors que le Fortran est destiné aux applications mathématiques, scientifiques et techniques, le Cobol est conçu pour informatiser les problèmes de gestion

en matière d'instructions de contrôle, il marque d'emblée les esprits en implémentant

- d'une part un 'vrai' (quoique rudimentaire) IF ... THEN ... ELSE ... (imbriquable),
- et d'autre part en créant les premiers états (pseudo)logiques : on peut ainsi associer à des variables scalaires (déclarées via 77) des (plusieurs si on veut !) états nommés (déclarés via 88) à valeur vrai/faux utilisables comme conditions

```

IDENTIFICATION DIVISION.
PROGRAM-ID.                                ALTERNATIVES.
*
*** declaration des donnees *****
*
DATA DIVISION.
WORKING-STORAGE SECTION.
*
* variables
*
77 SEXE                                     PIC X.
88 OKSEXE                                 VALUE 'H' 'h' 'F' 'f'.
88 HOMME                                 VALUE 'H' 'h'.
77 STATUT                                  PIC X.
88 OKSTATUT                              VALUE 'N' 'n' 'O' 'o'.
88 MARIEE                                VALUE 'O' 'o'.
*
*** sequence principale *****
*
PROCEDURE DIVISION.
MAIN SECTION.
    DISPLAY "quel est votre sexe (H/F) : ".
    ACCEPT SEXE.
    IF NOT OKSEXE
        DISPLAY "erreur sexe"
    ELSE
        IF HOMME
            DISPLAY "bonjour Monsieur "
        ELSE
            DISPLAY "etes-vous mariee (O/N) ? "
            ACCEPT STATUT
            IF NOT OKSTATUT
                DISPLAY "erreur etat civil"
            ELSE
                IF MARIEE
                    DISPLAY "bonjour Madame"
                ELSE
                    DISPLAY "bonjour Mademoiselle".
*
    EXIT PROGRAM.
    STOP RUN.
*
*****

```

## Pascal

on retrouve – forcément – les différentes possibilités évoquées en pseudo-code : variables logiques à part entière, instructions conditionnelle et alternative, raccourcis d'expressions logiques avec NOT, AND et OR grâce à l'opérateur IN

```

program alternatives1;
uses crt;
var sexe, statut : char;
begin
  clrscr;
  write('quel est le sexe (m/f) ? ');
  readln(sexe);
  if not (sexe in ['M', 'm', 'F', 'f']) then
    writeln('erreur sexe !')           { une seule instruction }
  else if sexe in ['M', 'm'] then
    writeln('bonjour Monsieur')       { une seule instruction }
  else begin                           { début séquence ! }
    write('etes-vous mariee (o/n) ? ');
    readln(statut);
    if not (statut in ['O', 'o', 'N', 'n']) then
      writeln('erreur etat-civil !')   { une seule instruction }
    else if statut in ['O', 'o'] then
      writeln('bonjour Madame')       { une seule instruction }
    else
      writeln('bonjour Mademoiselle')
    end;                               { fin séquence ! }
  writeln;
  readkey
end.

```

pour faire 'comme en Cobol', il faut créer des variables de type Logique (boolean), mais leur affectation est à charge du programmeur, donc – ici – pas d'un grand intérêt

```

program alternatives2;
uses crt;
var sexe, statut : char;
    sexeOk, estHomme, statutOk, estMariee : boolean;
begin
  clrscr;
  write('quel est le sexe (m/f) ? ');
  readln(sexe);
  sexeOk := sexe in ['M', 'm', 'F', 'f'];
  estHomme := sexe in ['M', 'm'];
  if not sexeOk then
    writeln('erreur sexe !')
  else if estHomme then
    writeln('bonjour Monsieur')
  else begin
    write('etes-vous mariee (o/n) ? ');
    readln(statut);
    statutOk := statut in ['O', 'o', 'N', 'n'];
    estMariee := statut in ['O', 'o'];
    if not statutOk then
      writeln('erreur etat-civil !')
    else if estMariee then
      writeln('bonjour Madame')
    else
      writeln('bonjour Mademoiselle')
    end;
  writeln;
  readkey
end.

```

c) STRUCTURES ALTERNATIVES MULTIPLES

Même réécrits de manière plus compacte, ces tests imbriqués restent lourds et d'une lecture souvent laborieuse, alors que la logique du problème devrait 'sauter aux yeux' à la lecture ! Nous allons à présent utiliser du 'sucre syntaxique'<sup>40</sup> pour soulager l'écriture tout en augmentant la lisibilité

➤ SelonQue ... valué

Appelée également choix multiple, (ou encore sélecteur de cas, aux écritures diverses mais équivalentes en pseudo-code : *ChoixParmi ...*, *DéciderEntre ...*) cette instruction de contrôle condense de manière élégante une batterie de *Si ... Alors ... Sinon ...* imbriqués; la syntaxe générale est la suivante :

```
SelonQue expression valuée [vaut | parmi | entre]
    valeurs : séquence1
    valeurs : séquence2
    [...]
    [Sinon : séquence]
finSQ
```

Par « expression valuée » dans le bloc de référence syntaxique ci-dessus, il faut comprendre :

toute expression qui renvoie une valeur scalaire (les valeurs littérales et les constantes n'ont aucun intérêt à cet endroit) ; il s'agit donc d'expressions contenant au moins une variable (ou une fonction)

et par « valeurs » ci-dessus, il faut comprendre :

une valeur littérale, une constante symbolique, un intervalle (p.ex. 10..99), une liste de valeurs (p.ex. 'O', 'o', 'N', 'n') ...

Attention : il faut comprendre le 'fonctionnement' de cette instruction comme suit :

- l'expression est évaluée et sa valeur est ensuite comparée successivement de haut en bas à chacun des cas listés :
- dès qu'il y a concordance, la séquence correspondante est exécutée et on quitte l'instruction<sup>41</sup> ;
- si aucune concordance n'est trouvée, la clause *Sinon* (facultative) est exécutée

```
Exemple :      Variable opérateur : Caractère
                nombre1, nombre2 : Réel

                ...
                lire(nombre1)
                lire(nombre2)
                lire(opérateur)

                SelonQue opérateur vaut
                '+' : écrire(nombre1 + nombre2)
                '-' : écrire(nombre1 - nombre2)
                '*' : écrire(nombre1 * nombre2)
                '/' : Si nombre2 <> 0 Alors
                        écrire(nombre1 / nombre2)
                    Sinon
                        écrire("erreur : division par 0")
                    finSi
                Sinon : écrire("erreur : opérateur inconnu !")
                finSQ
```

<sup>40</sup> [http://fr.wikipedia.org/wiki/Sucre\\_syntaxique](http://fr.wikipedia.org/wiki/Sucre_syntaxique) : bizarre d'utiliser du sucre pour faire 'maigrir' l'écriture d'un algorithme !

<sup>41</sup> en programmation, il faut soigneusement vérifier la documentation : Pascal et C diffèrent totalement sur ce point !

```

Exemple : Variable mois : 1..12
           moisDe30, moisDe31, moisDeFévrier : Logique
           ...
           lire(mois)
SelonQue mois
           parmi (1, 3, 5, 7, 8, 10, 12) : moisDe31 ← vrai
           parmi (4, 6, 9, 11) : moisDe30 ← vrai
           vaut 2 : moisDeFévrier ← vrai
           Sinon : écrire("erreur de mois")
finSQ

```

```

Exemple : Variable age : Entier
           ...
           lire(age)
SelonQue age
           entre 0 et 11 : écrire("enfant")
           entre 12 et 18 : écrire("adolescent")
           entre 20 et 60 : écrire("adulte")
           entre 60 et 80 : écrire("âgé")
           Sinon : écrire("vieux")
finSQ

```

```

Exemple : Variable jour : 1..7
           ...
           lire(jour)
SelonQue jour
           entre 1 et 5 : écrire("semaine")
           vaut 6 : écrire("samedi")
           vaut 7 : écrire("dimanche")
           Sinon : écrire("erreur de jour")
finSQ

```

Dans la mesure où il n'y a aucune ambiguïté, les différentes clauses peuvent donc être mélangées; ici encore c'est l'expressivité maximale qui est recherchée

La clause `Sinon` est facultative, mais on remarquera qu'elle est bien utile : elle permet au sélecteur de cas de contenir à la fois un validateur et un sélecteur de cas proprement dit

NB : le terminateur `finSQ` permet d'imbriquer des sélecteurs sans ambiguïté

```

Exemple : Variable sexe, statut : Caractère
           ...
           écrire("quel est le sexe (m/f) ? ")
           lire(sexe)
SelonQue sexe
           parmi ('M', 'm') : écrire("bonjour Monsieur")
           parmi ('F', 'f') :
               écrire("êtes-vous mariée (o/n) ? ")
               lire(statut)
               SelonQue statut
                   parmi ('O', 'o') : écrire("bonjour Madame")
                   parmi ('N', 'n') : écrire("bonjour Mademoiselle")
                   Sinon : écrire("erreur de statut")
               finSQ
           Sinon : écrire("erreur de sexe")
finSQ

```

## ➤ SelonQue ... conditionné

Dans cette version, ce n'est pas une 'expression évaluée' qui détermine laquelle des séquences listées sera exécutée, mais une liste de conditions (la première qui est vraie l'emporte ...) <sup>42</sup>

```

SelonQue
  condition1 : séquence1
  condition2 : séquence2
  [...]
[Sinon
  séquence]
finSQ

```

```

Exemple :  Variable nombre1, nombre2 : Entier
...
lire(nombre1); lire(nombre2)
SelonQue
  nombre1 > 0 : ...
  nombre2 = nombre1 : ...
  nombre2 < -3 : ...
Sinon : ...
finSQ

```

L'exemple habituel pour illustrer cette forme du selonQue est la détermination du nombre de racines d'une équation du second degré  $y = a.x^2 + b.x + c$  ; on sait que ce nombre (0, 1, 2) est déterminé par le signe du discriminant  $b^2 - 4a.c$  ( $< 0$  : aucune racine,  $= 0$  : une racine,  $> 0$  : deux racines) ; on écrirait donc :

```

Variable a, b, c, delta : Réel
...
lire(a); lire(b); lire(c)
delta ← b*b - 4*a*c
SelonQue
  delta > 0.0 : écrire("deux racines")
  delta = 0.0 : écrire("une racine")
  Sinon : écrire("aucune solution")
finSQ

```

On remarquera qu'il est possible de passer d'une forme à l'autre :

- ⇒ de la évaluée à la conditionnée, toujours, parce que la évaluée 'cache' en fait des comparaisons ;
- ⇒ de la conditionnée à la évaluée : pas toujours, p.ex. si les conditions portent sur des variables différentes ou si l'on ne sait pas énumérer les valeurs à tester, comme dans l'exemple précédent sur le second degré, ou si les valeurs sont réelles :

Exemple d'équivalence : (*évaluée à gauche, conditionnée à droite*)

```

variable jour : 1..7
...
lire(jour)
SelonQue mois
  entre 1 et 5 : écrire("semaine")
  vaut 6 : écrire("samedi")
  vaut 7 : écrire("dimanche")
  Sinon : écrire("erreur de jour")
finSQ

```

```

variable jour : 1..7
...
lire(jour)
SelonQue
  mois ≥ 1 et mois ≤ 5 : écrire("semaine")
  mois = 6 : écrire("samedi")
  mois = 7 : écrire("dimanche")
  Sinon : écrire("erreur de jour")
finSQ

```

<sup>42</sup> cette forme du selonQue n'est pas disponible dans les langages 'anciens' (Pascal, C, Java ...), par contre on la trouve depuis quasi toujours en VB @ ... elle refait heureusement surface dans les langages récents, Ruby, p.ex.

## ➤ Concrètement

### Fortran

il faut attendre les versions 'tardives' du langage (Fortran 90 : Fortran 95, ...) pour voir apparaître des instructions de contrôle 'par bloc' (IF ... ELSE ... END IF : *bye bye cher GOTO !*) et une forme multi-valuée du SelonQue (SELECT CASE ... END SELECT);

par contre il n'y a toujours pas de version multi-conditionnée de cette instruction (sinon en 'bidulant' de manière plus ou moins élégante ...)

```

PROGRAM ALTERNATIVE

c *** declaration des variables
      CHARACTER SEXE, ETAT
c *** formats d'entree/sortie
11     FORMAT('quel est le sexe (M/F) ? ')
12     FORMAT('etes-vous mariee (O/N) ?')
15     FORMAT('erreur sexe ! stop', 1X)
16     FORMAT('erreur etat-civil ! stop')
17     FORMAT('bonjour Monsieur')
18     FORMAT('bonjour Madame')
19     FORMAT('bonjour Mademoiselle')
21     FORMAT(1X)

c *** programme principal
      WRITE(6, 11)
      READ(5, 21) SEXE
c
      SELECT CASE (SEXE)
      CASE ('M', 'm')
        WRITE(6, 17)
      CASE ('F', 'f')
        WRITE(6, 12)
        READ(5, 21) ETAT
        SELECT CASE (ETAT)
        CASE ('O', 'o')
          WRITE(6, 18)
        CASE ('N', 'n')
          WRITE(6, 19)
        CASE DEFAULT
          WRITE(6, 16)
        END SELECT
      CASE DEFAULT
        WRITE(6, 15)
      END SELECT
c
      END

```

## Cobol

le Cobol s'adapte à la programmation structurée 'moderne' dès sa version majeure ANS 85 (instructions de contrôle avec terminateur explicite, ce qui réduit le nombre de paragraphes-procédures); un selonQue multi-valué apparaît, qui profite au maximum des états logiques (niveaux 88 de données) présents depuis toujours ...; classiquement, il utilise le mot `EVALUATE` et des clauses `WHEN`

```
IDENTIFICATION DIVISION.
PROGRAM-ID.                SELONQUE.
*
DATA DIVISION.
WORKING-STORAGE SECTION.
* variables
77 SEXE                     PIC X.
   88 FEMME                 VALUE 'F' 'f'.
   88 HOMME                 VALUE 'H' 'h'.
77 STATUT                   PIC X.
   88 MARIEE                VALUE 'O' 'o'.
   88 CELIB                 VALUE 'N', 'n'.
*
PROCEDURE DIVISION.
MAIN SECTION.
  DISPLAY "quel est votre sexe (H/F) ? ".
  ACCEPT SEXE.
  EVALUATE TRUE
    WHEN HOMME DISPLAY "bonjour Monsieur "
    WHEN FEMME
      DISPLAY "etes-vous mariee (O/N) ? "
      ACCEPT STATUT
      EVALUATE TRUE
        WHEN MARIEE DISPLAY "bonjour Madame"
        WHEN CELIB  DISPLAY "bonjour Mademoiselle"
        WHEN OTHER  DISPLAY "erreur etat civil"
      END-EVALUATE
    WHEN OTHER DISPLAY "erreur sexe"
  END-EVALUATE.
*
EXIT PROGRAM.
STOP RUN.
```

... dans la foulée, notons qu'une version multi-conditionnée est également proposée ...

```
IDENTIFICATION DIVISION.
PROGRAM-ID.                SELONQUE.
*
DATA DIVISION.
WORKING-STORAGE SECTION.
*
77 COEFA                    PIC 99V9.
77 COEFB                    PIC 99V9.
77 COEFC                    PIC 99V9.
77 DELTA                    PIC 9999V9.
*
PROCEDURE DIVISION.
MAIN SECTION.
  DISPLAY "introduisez le coefficient A : ".
  ACCEPT COEFA.
  DISPLAY "introduisez le coefficient B : ".
  ACCEPT COEFB.
  DISPLAY "introduisez le coefficient C : ".
  ACCEPT COEFC.
*
  COMPUTE DELTA = COEFB * COEFB - 4.0 * COEFA * COEFC.
*
  EVALUATE TRUE
    WHEN DELTA < 0.0 DISPLAY "aucune racine !"
    WHEN DELTA = 0.0 DISPLAY "deux racines confondues"
    WHEN OTHER      DISPLAY "deux racines distinctes"
  END-EVALUATE
*
EXIT PROGRAM.
STOP RUN.
```

fin des années (19)70, Pascal et C proposent chacun une implémentation (au comportement différent !) du selonQue multi-valué (n'acceptant l'un et l'autre que des données énumérables : entiers et caractères)

malheureusement, ni l'un ni l'autre ne proposeront jamais une version multi-conditionnée

### Pascal

```

program selonQue;
uses crt;
var sexe, statut : char;
begin
  clrscr;
  write('quel est le sexe (m/f) ? ');
  readln(sexe);
  case sexe of
    'M', 'm' : writeln('bonjour Monsieur');
    'F', 'f' : begin
      write('etes-vous mariee (o/n) ? ');
      readln(statut);
      case statut of
        'O', 'o' : writeln('bonjour Madame');
        'N', 'n' : writeln('bonjour Mademoiselle');
        else writeln('erreur etat-civil !')
      end end;
    else writeln('erreur sexe !')
  end;
  writeln;
  readkey
end.

```

### C

```

// exemple de selonQue valué
#include <stdio.h>           // librairie
#include <stdlib.h>          // librairie
// séquence principale
int main() {
  char sexe, statut;
  system("cls");
  printf("quel est le sexe (m/f) ? ");
  sexe = getchar();
  fflush(stdin);
  switch (sexe) {
    case 'M': case 'm':           // NB : exception : pas de { } ici
      printf("bonjour Monsieur\n");
      break;
    case 'F': case 'f':
      printf("etes-vous mariee (o/n) ? ");
      statut = getchar();
      fflush(stdin);
      switch (statut) {
        case 'O': case 'o' :
          printf("bonjour Madame\n");
          break;
        case 'N': case 'n':
          printf("bonjour Mademoiselle\n");
          break;
        default:
          printf("erreur etat-civil !\n");
          break;
      }
      break;
    default:
      printf("erreur de sexe !\n");
  }
  system("pause");
  return 0;
}

```



d) EXEMPLE RÉCAPITULATIF COMMENTÉ

On introduit la date du jour (numéro du jour de la semaine, le jour, le mois et l'année) et il faut écrire un algorithme qui détermine la date du lendemain (dans les mêmes termes) : interface du genre

```
introduisez la date du jour :
  année : 2008
  mois : 12
  jour : 31

quel est le numéro du jour de la semaine ? 7

aujourd'hui : dimanche 31 décembre 2008
demain : lundi 1 janvier 2009
```

*Pour rappel : il y a des mois de 31 jours (janvier, mars, mai, juillet, août, octobre, décembre), des mois de 30 jours (avril, juin, septembre, novembre) et le mois de février qui compte 28 ou 29 jours selon que l'année est bissextile ou non; une année est bissextile si elle est multiple de 4, sauf si elle séculaire (multiple de 100), mais elle l'est quand même si elle est multiple de 400 : [http://fr.wikipedia.org/wiki/Année\\_bissextile](http://fr.wikipedia.org/wiki/Année_bissextile)*

Dans un premier temps, on considèrera l'utilisateur comme idéal : aucune erreur à l'encodage; on reviendra ensuite sur les (indispensables) validations (et on écrira de véritables validateurs dans le chapitre consacré aux boucles)

On s'obligera toujours à écrire une version procédurale de l'algorithme, pour dégager les 'blocs fonctionnels' et les rendre les moins dépendants les uns des autres (ils se contentent de partager les mêmes données)

## ▪ étape 1 : déclaration des variables

```
Constante : annéeInf = 1900, annéeSup = 2040
Variable  numéroJour : 1 .. 7      # dans la semaine : 1 = lundi, ...
          jour : 1 .. 31           # selon le mois et le caractère bissextile
          mois : 1 .. 12
          année : annéeInf .. annéeSup # sous-type via constantes
          estBissextile : Logique
```

## ▪ étape 2 : algorithme abstrait : squelette général, indépendant des données

```
Algorithme jourLendemain :
  # ici le bloc déclaratif des données ci-dessus
Début
  encoderDonnées
  afficherAujourd'hui
  calculerDemain      # calculer la date de demain
  afficherDemain
Fin.
```

## ▪ étape 3 : concrétiser l'algorithme : donner du 'corps' à chacune des procédures :

## ° a) encodage des données, affectation 'externe' dans les variables globales déclarées en 1)

```
Procédure encoderDonnées :
Début
  écrire("introduisez la date du jour :")
  écrire("année : ")
  lire(année)          # saisie = affectation externe
  écrire("mois : ")
  lire(mois)           # idem ...
  écrire("jour : ")
  lire(jour)           # idem ...
  écrire("quel est le numéro du jour de la semaine ? ")
  lire(numéroJour)     # idem
Fin;
```

- ° b) afficher date du jour : on remarque qu'afficher la date du jour et celle du lendemain peuvent utiliser une même procédure si on utilise les mêmes variables (celles encodées) pour calculer la date du lendemain; c'est ce qui sera fait ici

```

Procédure afficherAujourd'hui :
Début
    écrire("Aujourd'hui : ")
    afficherDate                # procédure d'affichage
Fin;

Procédure afficherDemain :
Début
    écrire("Demain : ")
    afficherDate                # la même procédure d'affichage
Fin;

```

- ° c) la procédure afficherDate peut elle-même se décomposer en procédures plus 'petites', spécialisées dans l'affichage du libellé du jour et du mois ... jouons le jeu de cette décomposition (chacun utilisera les variables globales déclarées précédemment)

```

Procédure afficherJour :
Début
    selonQue numéroJour vaut    # variable globale numéroJour
        1 : écrire("lundi ")
        2 : écrire("mardi ")
            # etc. : on présume que le lecteur a compris le principe
        7 : écrire("dimanche ")
    finSQ
Fin;

Procédure afficherMois :
Début
    selonQue mois vaut          # variable globale mois
        1 : écrire("janvier ")
            # etc.: idem ...
        12 : écrire("décembre ")
    finSQ
Fin;

Procédure afficherDate :
Début
    afficherJour
    écrire(jour)
    afficherMois
    écrire(année)
Fin;

```

- étape 4 : le "cœur" du problème

```

Procédure calculerDemain :
Début
    # détermination bissextile ou non
    selonQue                    # !!!! l'ordre des cas est critique !!!!
        année mod 400 = 0 : estBissextile ← vrai
        année mod 100 = 0 : estBissextile ← faux
        année mod 4 = 0 : estBissextile ← vrai
        Sinon : estBissextile ← faux
    finSQ
    # demain, c'est un jour de plus dans la semaine et un jour de plus dans le mois
    numéroJour ← numéroJour + 1
    jour ← jour + 1
    # il va falloir gérer le risque de dépassement

```

```

# quel sera le mois de demain ?

selonQue mois
  parmi (1, 3, 5, 7, 8, 10, 12) :      # mois de 31
    Si jour > 31 Alors
      mois ← mois + 1
      jour ← 1
    finSi
  parmi (4, 6, 9, 11) :              # mois de 30
    Si jour > 30 Alors
      mois ← mois + 1
      jour ← 1
    finSi
  Sinon :                            # et février ?
    Si estBissextile Alors
      Si jour > 29 Alors
        mois ← mois + 1
        jour ← 1 finSi
      Sinon
        Si jour > 28 Alors
          mois ← mois + 1
          jour ← 1
        finSi
      finSi
    finSi
finSQ

# quelle sera l'année de demain ?
Si mois > 12 Alors
  année ← année + 1
  mois ← 1
finSi

# quel sera le jour de la semaine de demain ?
Si numéroJour > 7 Alors
  numéroJour ← 1
finSi
Fin.

```

#### ▪ améliorations possibles / variantes

```

# détermination bissextile ou non :
# pas sûr que ceci soit plus clair que le selonQue

  estBissextile ← (année mod 400 = 0 ou année mod 400 = 0)
                  et non (année mod 100 = 0)

# quelle sera l'année de demain ?

  année ← année + mois div 12
  mois ← mois mod 13

# quel sera le jour de demain ?

  numéroJour ← numéroJour mod 8

```

- étape finale : on met tout cela ensemble

```

Algorithme jourLendemain :
Constante annéeInf = 1900, annéeSup = 2040
Variable numéroJour : 1..7  # dans la semaine : 1=lundi, ...
        jour : 1..31        # selon le mois et le caractère bissextile
        mois : 1..12
        année : annéeInf..annéeSup
        estBissextile : Logique

```

**Procédure** encoderDonnées :

Début

```

    écrire("introduisez la date du jour :")
    écrire("  année : ")
    lire(année)
    écrire("  mois : ")
    lire(mois)
    écrire("  jour : ")
    lire(jour)
    écrire("quel est le numéro du jour de la semaine : ")
    lire(numéroJour)

```

Fin

**Procédure** afficherJour :

Début

```

    selonQue numéroJour vaut
        1 : écrire("lundi ")
        2 : écrire("mardi ")
        # etc ... on présume que le lecteur a compris le principe ...
        7 : écrire("dimanche ")
    finSQ

```

Fin

**Procédure** afficherMois :

Début

```

    selonQue mois vaut
        1 : écrire("janvier ")
        2 : écrire("février ")
        # etc ... idem ...
        12 : écrire("décembre ")
    finSQ

```

Fin;

**Procédure** afficherDate :

Début

```

    afficherJour
    écrire(jour)
    afficherMois
    écrire(mois)
    écrire(année)

```

Fin

**Procédure** afficherAujourd'hui :

Début

```

    écrire("Aujourd'hui : ")
    afficherDate  # procédure d'affichage

```

Fin;

**Procédure** afficherDemain :

Début

```

    écrire("Demain : ")
    afficherDate  # la même procédure d'affichage

```

Fin;

```

Procédure calculerDemain :
Début
# détermination bissextile ou non
selonQue
    année mod 400 = 0 : estBissextile ← vrai
    année mod 100 = 0 : estBissextile ← faux
    année mod 4 = 0 : estBissextile ← vrai
    sinon : estBissextile ← faux
finSQ
# demain, c'est un jour de plus dans la semaine et dans le mois
numéroJour ← numéroJour + 1
jour ← jour + 1
# quel sera le jour de la semaine de demain ?
Si numéroJour > 7 Alors numéroJour ← 1 finSi
# quel sera le mois de demain ?
selonQue mois
    parmi (1, 3, 5, 7, 8, 10, 12) :
        Si jour > 31 Alors
            mois ← mois + 1
            jour ← 1
        finSi
    parmi (4, 6, 9, 11) :
        Si jour > 30 Alors
            mois ← mois + 1
            jour ← 1
        finSi
    Sinon
        Si estBissextile Alors
            Si jour > 29 Alors
                mois ← mois + 1
                jour ← 1
            finSi
        Sinon
            Si jour > 28 Alors
                mois ← mois + 1
                jour ← 1
            finSi
        finSi
    finSi
finSQ
# quelle sera l'année de demain ?
Si mois > 12 Alors
    année ← année + 1
    mois ← 1
finSi
Fin

# -----

Début # algo abstrait
    encoderDonnées
    afficherAujourd'hui
    calculerDemain      # calculer la date de demain
    afficherDemain
Fin.

# =====

```

- ensuite : et si on validait l'encodage, avec sortie immédiate et message approprié en cas d'erreur ?
- ⇒ commençons par revoir l'algorithme abstrait; pour inclure la gestion d'erreur, nous lui ajoutons un état ! (et une variable permettant de déceler le cas d'erreur rencontré pour pouvoir afficher le message approprié)

```

Algorithme jourLendemain :
Constante : annéeInf = 1900, annéeSup : 2040
Variable  numéroJour : 1..7           # dans la semaine : 1=lundi, ...
           jour : 1..31                 # selon le mois et le caractère bissextile
           mois : 1..12
           année : annéeInf..annéeSup
           estBissextile : Logique
           estValide : Logique          # état de validité de l'encodage
           casErreur : 0..7           # pour mémoriser l'erreur commise

Début # algo abstrait : aucune référence aux données concrètes
  encoderDonnées          # encodage et détermination de l'état estValide
  Si estValide Alors      # tester état
    afficherAujourd'hui   # comme précédemment
    calculerDemain         # comme précédemment
    afficherDemain         # comme précédemment
  Sinon
    afficherErreur        # gestion d'erreur : message
  finSi
Fin.

```

- ⇒ que faut-il valider ? le numéro de jour(1..7), le mois (1..12), l'année (annéeInf..annéeSup) ... et le jour, bien sûr : on ne peut accepter que des valeurs compatibles avec le mois et le caractère bissextile ou non de l'année : tout ceci sera caché au sein de la procédure encoderDonnées

```

Procédure encoderDonnées :
Début
  casErreur ← 0           # jusqu'ici tout va bien ! ☺
  écrire("date : année : ")
  lire(année)
  Si non (année entre annéeInf et annéeSup) Alors
    casErreur ← 1
  Sinon
    # détermination bissextile ou non
    # !!! à remarquer : la détermination de cet état a été déplacée ici
    selonQue
      année mod 400 = 0 : estBissextile ← vrai
      année mod 100 = 0 : estBissextile ← faux
      année mod 4 = 0 : estBissextile ← vrai
      sinon : estBissextile ← faux
    finSQ
  écrire("date : mois : ")
  lire(mois)
  Si non (mois entre 1 et 12) Alors
    casErreur ← 2
  Sinon
    écrire("date : jour : ")
    lire(jour)
    Si non (jour entre 1 et 31) Alors
      casErreur ← 3
    Sinon
      selonQue
        mois parmi (4, 6, 9, 11) et jour > 30 : casErreur ← 4
        mois = 2 et estBissextile et jour > 29 : casErreur ← 5
        mois = 2 et non estBissextile et jour > 28 : casErreur ← 6
      Sinon
        écrire("numéro du jour de la semaine : ")
        lire(numéroJour)
        Si non numéroJour entre 1 et 7 Alors casErreur ← 7
      finSQ
    finSi
  finSi
  finSi
  estValide ← (casErreur = 0)          # déterminer l'état final de l'encodage
Fin

```

quant à la procédure `afficherErreur`, un simple `selonQue` et le tour est joué !

**Procédure** `afficherErreur` :

Début

`selonQue casErreur vaut`

```
1 : écrire("année hors limite")
2 : écrire("mois hors limite")
3 : écrire("jour hors limite")
4 : écrire("jour incompatible avec mois de 30 jours")
5 : écrire("jour incompatible avec février bissextile")
6 : écrire("jour incompatible avec février non bissextile")
7 : écrire("jour de la semaine hors limite")
```

`finSQ`

`écrire("arrêt suite à cette erreur !")`

Fin

remarque finale : dans la procédure `calculerDemain`, on peut enlever le bloc qui détermine si l'année est bissextile, puisque le travail a été fait dans la procédure `encoderDonnées` (on fera mieux plus tard)

Étant donné qu'un cours de langage Pascal est donné en parallèle au cours de Principes de Programmation, voici une implémentation complète de ce problème en Pascal

*le lecteur sera particulièrement attentif à la manière dont sont traduites les différentes formes alternatives, à l'absence d'un équivalent du `SelonQue` multiconditionné et aux contraintes liées à l'absence de terminateurs d'instruction (maîtrise de l'imbrication des `begin...end`)*

```
program jourLendemain;
uses crt;
const anneeInf = 1900; anneeSup = 2040;
var  numeroJour : 1..7; { dans la semaine : 1=lundi, ... }
    jour : 1..31;      { selon le mois et le caractère bissextile }
    mois : 1..12;
    annee : anneeInf..anneeSup;
    estBissextile : boolean;
    estValide : boolean; { état de validité de l'encodage }
    casErreur : 0..7;    { pour mémoriser l'erreur commise }

procedure encoderDonnees;
begin
    casErreur := 0;
    writeln('introduisez la date du jour :');
    write('  annee : '); readln(annee);
    if not (annee in [anneeInf..anneeSup]) then casErreur := 1
    else begin
        { détermination bissextile ou non }
        if annee mod 400 = 0 then estBissextile := true
        else if annee mod 100 = 0 then estBissextile := false
        else if annee mod 4 = 0 then estBissextile := true
        else estBissextile := false;
        write('  mois : '); readln(mois);
        if not (mois in [1..12]) then casErreur := 2
        else begin
            write('    jour : '); readln(jour);
            if not (jour in [1..31]) then casErreur := 3
            else begin
                case mois of
                    4, 6, 9, 11 : if jour > 30 then casErreur := 4;
                    2 : if estBissextile and (jour > 29) then casErreur := 5
                        else if not estBissextile and (jour > 28) then casErreur := 6
                        else begin
                            write('quel est le numero du jour de la semaine : ');
                            readln(numeroJour);
                            if not (numeroJour in [1..7]) then casErreur := 7
                            end
                        end
                end
            end
        end
    end
end;
estValide := (casErreur = 0)
end;
```

```

procedure afficherJour;
begin
  case numeroJour of
    1 : write('lundi ');
    2 : write('mardi ');
    3 : write('mercredi ');
    4 : write('jeudi ');
    5 : write('vendredi ');
    6 : write('samedi ');
    7 : write('dimanche ')
  end
end;

procedure afficherMois;
begin
  case mois of
    1 : write('janvier ');
    2 : write('fevrier ');
    3 : write('mars ');
    4 : write('avril ');
    5 : write('mai ');
    6 : write('juin ');
    7 : write('juillet ');
    8 : write('aout ');
    9 : write('septembre ');
    10 : write('octobre ');
    11 : write('novembre ');
    12 : write('decembre ')
  end
end;

procedure afficherDate;
begin
  afficherJour;
  write(jour);
  write(' ');
  afficherMois;
  writeln(annee)
end;

procedure afficherAujourd'hui;
begin
  write('Aujourd'hui : ');
  afficherDate { procédure d'affichage }
end;

procedure afficherDemain;
begin
  write('Demain : ');
  afficherDate { la même procédure d'affichage }
end;

procedure calculerDemain;
begin
  { demain, c'est un jour de plus dans la semaine et dans le mois }
  numeroJour := numeroJour + 1;
  jour := jour + 1;
  { quel sera le jour de la semaine de demain ? }
  if numeroJour > 7 then numeroJour := 1;
  { quel sera le mois de demain ? }
  case mois of
    1, 3, 5, 7, 8, 10, 12 : if jour > 31 then begin
                           mois := mois + 1;
                           jour := 1
                         end;
    4, 6, 9, 11 : if jour > 30 then begin
                  mois := mois + 1;
                  jour := 1
                end;
    else if estBissextile then if jour > 29 then begin
                              mois := mois + 1;
                              jour := 1 end
    else if jour > 28 then begin
                              mois := mois + 1;
                              jour := 1 end
  end;
end;

```



```

    { quelle sera l'année de demain ? }
    if mois > 12 then begin
        annee := annee + 1;
        mois := 1
    end
end;

procedure afficherErreur;
begin
    case casErreur of
        1 : writeln('annee hors limite');
        2 : writeln('mois hors limite');
        3 : writeln('jour hors limite');
        4 : writeln('jour incompatible avec mois de 30 jours');
        5 : writeln('jour incompatible avec fevrier bissextile');
        6 : writeln('jour incompatible avec fevrier non bissextile');
        7 : writeln('jour de la semaine hors limite')
    end;
    writeln('arret suite a cette erreur !')
end;

{ ----- }

begin { algo abstrait }
    encoderDonnees;
    if estValide then begin
        afficherAujourd'hui;
        calculerDemain;      { calculer la date de demain }
        afficherDemain end
    else
        afficherErreur;
        readkey
    end.
{ ===== }

```

e) LES TABLES DE DÉCISION➤ Généralités

Lorsque l'exécution d'actions dépend de plusieurs états, il convient de les évaluer ensemble et non séparément

- soit en utilisant des conditions plus complexes (expressions logiques) faisant appel aux opérateurs NON, ET, OU
- soit en imbriquant des structures alternatives binaires (SI ... ALORS ... SINON ...), multi-valuées ou multi-conditionnées (SELON QUE ...)

La tâche est relativement aisée lorsque ces états sont peu nombreux, mais devient plus complexe avec la multiplication des conditions et de leurs couplages ... et des questions importantes apparaissent :

- comment vérifier que tous les cas ont été pris en compte (complétude) ?
- comment vérifier qu'il n'y a pas de contradictions ?
- est-il possible de trouver des simplifications (minimalité) ?
- comment finalement traduire au mieux l'ensemble de manière algorithmique ?

Les Tables de décision constituent une solution technique intéressante préalable à la rédaction de l'algorithme : elles représentent sous forme de tableau l'ensemble des conditions (états) et l'ensemble des actions qui en dépendent; cette forme compacte rend l'examen, la validation et la simplification de l'ensemble bien plus facile

La structure générale d'une Table de décision est la suivante

Conditions	valeurs
Actions	décisions

➤ Construction

Pour expliquer la démarche pas à pas, nous allons partir d'un exemple simpl(ist)e : automatiser la partie salutation d'une lettre (c.-à-d. générer automatiquement le titre : Monsieur, Madame ou Mademoiselle)

- bien entendu, le point de départ est entièrement à charge de l'analyste : c'est à lui de déterminer que le titre à utiliser dépend du couplage de deux états multivalués : le sexe (Masculin/Féminin) et l'état-civil (Célibataire/Marié/Séparé/Divorcé/Veuf)
- on construit alors la partie supérieure du tableau avec les états et en ayant soin d'exprimer toutes les combinaisons (2 possibilités pour sexe \* 5 possibilités pour état-civil = 10 colonnes)

conditions	valeurs									
sexe	M	M	M	M	M	F	F	F	F	F
état-civil	C	M	S	D	V	C	M	S	D	V

il est à remarquer que cette construction génère des couples de valeurs (sexe/état-civil) tous différents ! (ce sera une des premières vérifications à effectuer)

- ensuite on place les différentes actions (ici les titres à utiliser) dans la première colonne (actions) de la partie inférieure

conditions	valeurs									
sexe	M	M	M	M	M	F	F	F	F	F
état-civil	C	M	S	D	V	C	M	S	D	V
actions										
Monsieur										
Mademoiselle										
Madame										

- enfin, on met (par exemple) une croix dans la partie inférieure droite (décisions) pour indiquer quelle(s) condition(s) entraîne(nt) quelle(s) action(s)

conditions	valeurs									
sexe (M, F)	M	M	M	M	M	F	F	F	F	F
état-civil (C, M, S, D, V)	C	M	S	D	V	C	M	S	D	V
actions	décisions									
Monsieur	X	X	X	X	X					
Mademoiselle						X				
Madame							X	X	X	X

ici, il faut une et une seule croix dans chaque colonne, et au moins une croix dans chaque ligne, cependant, il peut y avoir plusieurs croix sur une même ligne

À ce stade, on peut facilement prendre contact avec le 'client' de l'application pour revoir avec lui si le problème a été bien compris, est complet et ne contient pas de contradiction

### ➤ Alternatives

On sait à présent que des états multivalués (comme ici) peuvent aisément se ramener à un ensemble équivalent d'états binaires (vrai/faux) via des comparaisons sur leurs valeurs

P.ex. la première colonne correspond à l'expression logique :  $\text{sexe} = \text{'M'}$  ET  $\text{état-civil} = \text{'C'}$

Cependant, on évitera autant que possible une version entièrement binaire (les valeurs ne sont alors plus que des Vrai/faux) des Tables de décision, par exemple

conditions	valeurs									
sexe = 'M'	V	V	V	V	V	F	F	F	F	F
état-civil = 'C'	V	F	F	F	F	V	F	F	F	F
état-civil = 'M'	F	V	F	F	F	F	V	F	F	F
état-civil = 'S'	F	F	V	F	F	F	F	V	F	F
état-civil = 'D'	F	F	F	V	F	F	F	F	V	F
état-civil = 'V'	F	F	F	F	V	F	F	F	F	V
actions	décisions									
Monsieur	X	X	X	X	X					
Mademoiselle						X				
Madame							X	X	X	X

c'est nettement plus lourd, moins lisible et la possibilité d'introduire des contradictions est nettement plus risquée, p.ex.

conditions	valeurs									
sexe = 'M'	V	V	V	V	V	F	F	F	F	F
état-civil = 'C'	V	F	F	V	F	V	F	F	F	F
état-civil = 'M'	V	V	F	F	F	F	V	F	F	F

### ➤ Simplifications

Ainsi les "Tables de décisions" peuvent-elles être ramenées au concept (en Logique mathématique<sup>43</sup>) de "Table de vérité" sur lesquelles les notions de Formes canoniques et de Simplifications sont bien connues et maîtrisées

Avez-vous remarqué, le tableau final (au-dessus de la page) est une version 'Forme canonique "somme de produits"' de la Table de décision : toutes les possibilités y sont présentes et constituées d'expressions connectant valeurs de sexe et d'état-civil par l'opérateur ET

on peut donc écrire l'ensemble du tableau par  $(\text{sexe}=\text{'M' ET état-civil}=\text{'C'}) \text{ OU } (\text{sexe}=\text{'C' ET état-civil}=\text{'M'}) \text{ OU } \dots \text{ etc } \dots$

À présent, comme en Logique, on va chercher à simplifier le tableau

<sup>43</sup> faut-il le préciser ? la (re)lecture du chapitre correspondant du Cours de Mathématiques est vivement conseillée ...

3 lignes d'actions pour 10 colonnes de conditions : cela signifie que certaines lignes comportent plusieurs croix et qu'il est envisageable d'effectuer des regroupements (comme les 'mises en évidence' en Logique)

conditions	valeurs									
sexe (M, F)	M	M	M	M	M	F	F	F	F	F
état-civil (C, M, S, D, V)	C	M	S	D	V	C	M	S	D	V
actions	décisions									
Monsieur	X	X	X	X	X					
Mademoiselle						X				
Madame							X	X	X	X

- la première ligne indique que le titre 'Monsieur' ne dépend que de la seule condition sexe='M' et en aucun cas de l'état-civil : on ne conserve qu'une seule colonne sur les cinq en indiquant par un '-' que l'état-civil est indifférent (n'intervient pas)
- les deux autres lignes (sexe='F') montrent une sensibilité binaire à l'état-civil (Célibataire=>Mademoiselle et les autres cas =>Madame) : on ne conserve que deux colonnes, une avec le couple de valeurs (F, C) et l'autre avec le couple de valeurs (F, -)
- on simplifiera donc le tableau comme suit :

conditions	valeurs		
sexe (M, F)	M	F	F
état-civil (C, M, S, D, V)	-	C	-
actions	décisions		
Monsieur	X		
Mademoiselle		X	
Madame			X

- et on rédigera finalement la partie conditionnelle de l'algorithme :

```

Variable sexe, étatCivile : Caractère,
        titre : Texte

...
lire(sexe)
lire(étatCivile)
...
Si sexe = 'M' Alors
    titre ← "Monsieur"
Sinon                                     # donc une femme
    Si étatCivile = 'C' Alors
        titre ← "Mademoiselle"
    Sinon
        titre ← "Madame"
    finSi
finSi
...

```

- ou de manière équivalente (plus compacte, mais peut-être moins claire en première lecture)

```

Variable sexe, étatCivile : Caractère,
        titre : Texte

...
lire(sexe)
lire(étatCivile)
...
SelonQue
    sexe = 'M'           : titre ← "Monsieur"
    étatCivile = 'C'     : titre ← "Mademoiselle"
    Sinon               : titre ← "Madame"
finSQ
...

```

Si les états sont déterminés à partir de variables faisant l'objet d'un encodage externe, il peut être intéressant (il est nécessaire, plutôt) de prévoir les validations dans le processus d'analyse, il suffit d'ajouter une colonne supplémentaire par état (correspondant à "toute autre valeur que celles prévues") et une (ou plusieurs) ligne(s) d'action (correspondant à "que faire en cas d'encodage non conforme ?")

conditions	valeurs											
sexe (M, F)	M	M	M	M	M	F	F	F	F	F	autre	-
état-civil (C, M, S, D, V)	C	M	S	D	V	C	M	S	D	V	-	autre
actions	décisions											
Monsieur	X	X	X	X	X							
Mademoiselle						X						
Madame							X	X	X	X		
erreur											X	X

lors de la simplification, il suffira lors des regroupements de colonnes de spécifier la liste des valeurs (p.ex. pour l'état-civil des femmes)

conditions	valeurs				
sexe (M, F)	M	F	F	autre	-
état-civil (C, M, S, D, V)	-	C	M, S, D, V	-	autre
actions	décisions				
Monsieur	X				
Mademoiselle		X			
Madame			X		
erreur				X	X

et il y a plusieurs écritures équivalentes possibles de l'algorithme

```

Variable sexe, étatCivil : Caractère,
        titre : Texte, erreur : Logique
...
lire(sexe)
lire(étatCivil)
...
erreur ← faux
SelonQue
    sexe = 'M'           : titre ← "Monsieur"
    sexe = 'F'
        SelonQue
            étatCivil = 'C'           : titre ← "Mademoiselle"
            étatCivil parmi ('M', 'S', 'D', 'V') : titre ← "Madame"
            Sinon                   : erreur ← vrai
        finSQ
    Sinon : erreur ← vrai
finSQ
...

```

### ➤ un exemple

il a été conseillé dans l'exemple précédent de préférer une présentation multivaluée à une présentation 'tout binaire'; ceci ne veut pas dire que l'on ne peut pas utiliser d'état binaire ! dans bien des cas, beaucoup d'états (conditions) sont 'naturellement' binaires ... exemple(s) ...

*gérer les remboursements de mutuelle pour des visites médicales (cas fictif)*

*règles de base : pour qu'il y ait remboursement, il faut d'abord qu'un montant plancher ait été payé par le patient (Vrai/Faux), il y a alors remboursement à 50% pour une consultation à domicile (D), 65% pour une consultation au cabinet (C) du médecin et à 80% pour une consultation à l'hôpital (H) à condition que le médecin soit affilié (Vrai/Faux) à l'hôpital; dans le cas contraire, le taux de remboursement est celui d'une consultation au cabinet*

*construire la table complète :  $2 * 3 * 2 = 12$  colonnes*

conditions	valeurs										
montant plancher payé (V, F)	V	V	V	V	V	V	F	F	F	F	F
type de visite(D, C, H)	D	D	C	C	H	H	D	D	C	C	H
médecin affilié (V, F)	V	F	V	F	V	F	V	F	V	F	V
actions	décisions										
rembourser 50%	X	X									
rembourser 65%			X	X		X					
rembourser 80%					X						
pas de remboursement							X	X	X	X	X

*première simplification : pas de remboursement si pas de montant plancher*

conditions	valeurs						
montant plancher payé (V, F)	V	V	V	V	V	V	F
type de visite(D, C, H)	D	D	C	C	H	H	-
médecin affilié (V, F)	V	F	V	F	V	F	-
actions	décisions						
rembourser 50%	X	X					
rembourser 65%			X	X		X	
rembourser 80%					X		
pas de remboursement							X

*deuxième simplification : l'affiliation du médecin : uniquement pour consultation à l'hôpital*

conditions	valeurs				
montant plancher payé (V, F)	V	V	V	V	F
type de visite(D, C, H)	D	C	H	H	-
médecin affilié (V, F)	-	-	V	F	-
actions	décisions				
rembourser 50%	X				
rembourser 65%		X		X	
rembourser 80%			X		
pas de remboursement					X

*troisième simplification : les différentes actions peuvent s'exprimer à l'aide d'une seule variable numérique (taux de remboursement)*

conditions	valeurs				
montant plancher payé (V, F)	V	V	V	V	F
type de visite(D, C, H)	D	C	H	H	-
médecin affilié (V, F)	-	-	V	F	-
actions	décisions				
taux de remboursement	50%	65%	80%	65%	0%

*algorithme :*

```

Constante seuilMontant = ...           # à partir duquel remboursement
Variable typeVisite, affilié : Caractère,
        montant, tauxRemboursement : Réel
...
lire(montant)
lire(typeVisite)      # valider liste ('D', 'C', 'H')
lire(affilié)         # valider liste ('V', 'F')
...
tauxRemboursement ← 0.0
Si montant >= seuilMontant Alors
  selonQue typeVisite vaut
    'D' : tauxRemboursement ← 0.5
    'C' : tauxRemboursement ← 0.65
  Sinon
    Si affilié = 'V' Alors
      tauxRemboursement ← 0.8
    Sinon
      tauxRemboursement ← 0.65
    finSi
  finSq
finSi
...

```

*si l'on veut valider l'encodage des données de type Caractère, on ajoute une colonne pour chacun de ces états et une ligne d'action spécifique*

conditions	valeurs						
montant plancher payé (V, F)	V	V	V	V	F	-	-
type de visite(D, C, H)	D	C	H	H	-	autre	
médecin affilié (V, F)	-	-	V	F	-		autre
actions	décisions						
taux de remboursement	50%	65%	80%	65%	0%		
erreur						X	X

*on pourrait ainsi rassembler les validations en une seule ligne et gérer l'erreur via une variable Logique*

```

erreur ← faux
Si affilié parmi ('V', 'F') et typeVisite parmi ('D', 'C', 'H') Alors
  tauxRemboursement ← 0.0
  Si montant >= seuilMontant Alors
    selonQue typeVisite vaut
      'D' : tauxRemboursement ← 0.5
      'C' : tauxRemboursement ← 0.65
    Sinon
      Si affilié = 'V' Alors
        tauxRemboursement ← 0.8
      Sinon
        tauxRemboursement ← 0.65
      finSi
    finSq
  finSi
Sinon
  erreur ← vrai
finSi

```

### ➤ un autre exemple

à côté des cas multivalués, il existe d'autres types de conditions (états binaires purs, intervalles numériques, ...) mais la technique reste fondamentalement la même : exemple<sup>44</sup> ...

*Lise, la directrice des finances, doit décider dans quels cas elle exigera un paiement préalable des clients et quelles seront les différentes modalités de crédit.*

1. Elle identifie d'abord les conditions influant sur le crédit de ses clients. Elle sait, par exemple, que les clients du secteur public ont besoin d'un délai de paiement plus long. Elle pense aussi se baser sur les antécédents de paiement et sur le montant de la commande.
2. Elle envisage quatre types d'autorisations de crédit ou d'actions : paiement à l'avance, dépôt de 20%, délai de 30 jours, délai de 60 jours.
3. Le secteur d'origine du client a deux valeurs (public ou privé). Elle situerait les antécédents de paiement sur deux niveaux en fonction du paiement de la dernière commande (à temps ou en retard) et le montant de la commande à soit moins de 50,000€ soit 50,000€ ou plus.
4. Elle construit sa table de décision sur base d'états/conditions binaires (Vrai/Faux) et d'une condition sur un intervalle numérique

conditions	valeurs							
client du service public	V	V	V	V	F	F	F	F
paiement à temps	V	V	F	F	V	V	F	F
commande	< 50000	≥ 50000	< 50000	≥ 50000	< 50000	≥ 50000	< 50000	≥ 50000
actions	décisions							
paiement à l'avance								X
dépôt de 20%							X	
délai de 30 jours						X		
délai de 60 jours	X	X	X	X	X			

5. Elle vérifie la table avec des gens qui connaissent bien les pratiques de crédit dans le domaine. On lui dit que, dans les faits, le délai de 30 jours est standard dans le secteur privé.
6. Elle revoit sa première table et voit que les colonnes 1 à 4 peuvent être fusionnées. « Si » le client est du secteur public, « alors » il a toujours droit à un délai de 60 jours. D'autre part, l'intervalle peut se ramener à une condition simple. La table devient

conditions	valeurs				
client du service public	V	F	F	F	F
paiement à temps	-	V	V	F	F
commande < 50000€	-	V	F	V	F
actions	décisions				
paiement à l'avance					X
dépôt de 20%				X	
délai de 30 jours			X		
délai de 60 jours	X	X			

<sup>44</sup> d'après un exemple trouvé sur <http://www.er.ugam.ca>



### ➤ Concrètement

Bien souvent, lors de l'analyse d'un problème complexe, on arrive à un stade où "un tableau bien fait vaut mieux qu'un long discours ..."; d'où cette intéressante technique très visuelle de la Table de décision

Malheureusement, pour traduire une Table de décision, la plupart des langages de programmation (et même le pseudo-code) n'offrent habituellement que les outils 'de base' : une forme équivalente du Si ... Alors ... Sinon ... (IF ... THEN ...ELSE ...), parfois une version du selonQue multivalué (CASE en Pascal, SWITCH en C, Java, ...) et (beaucoup trop) rarement une version du selonQue multiconditionné (VB, Ruby, ...).

Il est donc difficile (sinon impossible) de retrouver la table de décision au premier coup d'œil en lisant le code ! (idéalement, cette table devrait donc se retrouver telle quelle sous forme de commentaires)

Il y a une célèbre exception : le langage Cobol auquel on a reproché – à juste titre bien souvent – d'être verbeux, de proposer des instructions qui sont de véritables 'usines' et qui sont le cauchemar du programmeur débutant ☹

par exemple (parmi bien d'autres), l'instruction EVALUATE : elle offre au programmeur le selonQue multi-valué, le selonQue multiconditionné et une implémentation directe de la Table de décision; sa syntaxe complète figure ci-dessous<sup>45</sup> :

```
EVALUATE { IDENT-1 | LITT-1 | EXPR-1 | TRUE | FALSE }
[ ALSO { IDENT-2 | LITT-2 | EXPR-2 | TRUE | FALSE } ] ...
{ { WHEN { ANY | COND-1 | TRUE | FALSE |
      NOT { { IDENT-3 | LITT-3 | EXPR-3 }
            [ THRU { IDENT-4 | LITT-4 | EXPR-4 } ]
          }
    ALSO [ { ANY | COND-2 | TRUE | FALSE |
            NOT { { IDENT-5 | LITT-5 | EXPR-5 }
                  [ THRU { IDENT-6 | LITT-6 | EXPR-6 } ]
                }
          }
    ] ...
  } ...
  IMPERATIVE-STATEMENT-1
} ...
[ WHEN OTHER IMPERATIVE-STATEMENT-2 ]
[ END-EVALUATE ]
```

illustrons la dernière possibilité par un exemple simple :

un exemple (fictif) dans le mode de l'assurance : il s'agit d'exprimer – en fonction de l'âge du conducteur, de la cylindrée et du nombre d'accidents déjà comptabilisés – quelle(s) règle(s) il faut appliquer pour déterminer la prime d'assurance à demander ... on dresse d'abord la Table de décision suivante ...

CONDITIONS	VALEURS
âge > 21	V V V V F F F
cylindrée > 2000	V V F F V F F
nombre d'accidents > 2	V F V F - V F
ACTIONS	DÉCISIONS
augmenter la prime de 45%	X - - - - -
augmenter la prime de 15%	- X - - - -
augmenter la prime de 30%	- - X - - -
prime inchangée	- - - X - -
supprimer l'assurance	- - - - X -
augmenter la prime de 50%	- - - - - X
augmenter la prime de 10%	- - - - - X

<sup>45</sup> extrait du manuel de référence du Cobol ANS85 de MicroFocus

... et ensuite, l'instruction EVALUATE de Cobol permet de traiter l'ensemble d'un seul coup (essayez d'en faire autant avec votre langage favori !) :

```
EVALUATE AGE > 21 ALSO CYLINDREE > 2000 ALSO ACCIDENTS > 2
  WHEN      TRUE      ALSO TRUE      ALSO TRUE      MULTIPLY PRIME BY 1.45
  WHEN      TRUE      ALSO TRUE      ALSO FALSE     MULTIPLY PRIME BY 1.15
  WHEN      TRUE      ALSO FALSE     ALSO TRUE      MULTIPLY PRIME BY 1.30
  WHEN      TRUE      ALSO FALSE     ALSO FALSE     CONTINUE
  WHEN      FALSE     ALSO TRUE      ALSO TRUE      SET EXCLU TO TRUE
  WHEN      FALSE     ALSO TRUE      ALSO FALSE     MULTIPLY PRIME BY 1.60
  WHEN      FALSE     ALSO FALSE     ALSO TRUE      MULTIPLY PRIME BY 1.50
  WHEN      FALSE     ALSO FALSE     ALSO FALSE     MULTIPLY PRIME BY 1.10
END-EVALUATE
```

convainquant, non ?

f) EXERCICES

*il est évidemment conseillé – lorsque l'exercice s'y prête – de réaliser une Table de décision et d'en rédiger l'algorithme ensuite ...*

1. Une compagnie d'assurance auto propose quatre types de tarifs A, B, C (du moins cher au plus cher) selon la situation du conducteur :

- un conducteur de moins de 25 ans, titulaire du permis depuis moins de deux ans aura un tarif C s'il n'a jamais été responsable d'accident ; dans le cas contraire, la compagnie refusera l'assurance
- un conducteur de moins de 25 ans et titulaire du permis depuis plus de deux ans, ou de plus de 25 ans mais titulaire du permis depuis moins de deux ans aura le tarif B s'il n'a jamais provoqué d'accident, le tarif C en cas d'un seul accident et ne sera pas assuré au-delà
- un conducteur de plus de 25 ans, titulaire du permis depuis plus de deux ans bénéficie du tarif A s'il n'est à l'origine d'aucun accident, du tarif B pour un seul accident, du tarif C pour deux accidents ; au-delà il ne sera pas assuré

rédiger un algorithme procédural comportant la déclaration et la saisie des données nécessaires au traitement de ce problème

2. Automate bancaire : étant donné le solde du compte du client et le montant qu'il souhaite retirer, écrire un algorithme procédural qui accepte (au clavier) le montant du solde et le montant du retrait, et indique – sachant que le plancher du compte de peut dépasser -1,000€ - si le retrait est autorisé, est refusé, ou le cas échéant, le montant du retrait partiel autorisé ; dans tous les cas, afficher le solde du compte après opération

3. Baby sitting

Pour les parents noctambules, une étudiante propose ses services selon le tarif suivant :

- 7.00€ l'heure, entre 17h00 et 21h59
- 9.00€ l'heure, entre 22h00 et 23h59
- 12.00€ l'heure, au-delà de minuit

Établir un algorithme procédural de calcul de la rémunération en respectant les contraintes supplémentaires suivantes :

- Validation à la saisie (introduction au clavier) :
  - heure de début (valide si entre 17 et 20) et minutes de début (valide si entre 0 et 59)
  - heure de fin (postérieure à l'heure de début)
- Calcul :
  - toute heure commencée est intégralement due

4. Anniversaire

À partir de votre date de naissance, et connaissant le jour de la semaine auquel cet heureux événement a eu lieu, quels sont (seront) les années de votre vie (on suppose que vous vivrez cent ans) où votre anniversaire tombe un dimanche ?

- pour déterminer le jour de la semaine de votre naissance, allez p.ex. sur [http://home.scarlet.be/~tor-4132/cal\\_tres\\_grand.htm](http://home.scarlet.be/~tor-4132/cal_tres_grand.htm) ou <http://actu63.free.fr/perpetuel.htm>
- on considère que le dimanche est le septième jour de la semaine

5. Pour pouvoir accéder à un poste de secrétaire aux Communautés européennes, il faut avoir moins de 40 ans, parler au moins trois langues et avoir un bac économique ou un bac en secrétariat ou une expérience d'au moins 3 ans; pour un poste de responsable administratif, il faut avoir moins de 50 ans, parler au moins trois langues, avoir un diplôme universitaire et une expérience d'au moins 5 ans

Quelles sont les données (constantes et variables) nécessaires, rédiger l'algorithme sur base d'une Table de décision

6. Suite à des difficultés économiques, une entreprise décide de "dégraisser" son personnel; le plan suivant est présenté au Conseil d'entreprise : les employés d'au moins 55 ans possédant au moins 10 ans d'ancienneté dans l'entreprise recevront une 'prime' de licenciement équivalente à 200% de leur dernier salaire; il en va de même pour les employés de moins de 60 ans avec une ancienneté d'au moins 5 ans; les employés qui ont entre 55 et 60 ans et une ancienneté entre 5 et 10 ans recevront une 'prime' équivalente à 150% de leur dernier salaire

Rédiger l'algorithme établissant le montant de la prime sur base de l'âge et de l'ancienneté supposées valides

7. Julien veut préparer son sac à dos pour partir en randonnée<sup>46</sup>

Selon les conditions, il doit emporter plus ou moins de matériel.

- si la température est inférieure à 15 degrés, il doit prendre des vêtements supplémentaires, qui pèsent au total 1.5 kg.
- si la pluie est annoncée, il emporte une cape de pluie de 0.5 kg, sauf si la température est supérieure à 25 degrés.
- il prend une gourde (pleine) de 1 kg, et si la température dépasse 30 degrés, il en prend une deuxième.
- si son ami Max l'accompagne, c'est Max qui apporte le casse-croûte, sinon Julien doit porter 0.4 kg de plus.
- si la pluie n'est pas annoncée, Julien prolongera un peu la randonnée, il prend donc 0.2 kg d'aliments supplémentaires (sauf si c'est Max qui apporte le casse-croûte).

Au moyen d'une Table de décision, rédigez un algorithme concret qui calcule le poids du sac à dos

- déclarez-y les constantes qui vous paraissent appropriées
- déclarez-y les variables du problème; pour chacune (température, présence de Max, pluie annoncée, poids du sac), choisissez le type de donnée le plus adapté à sa représentation (Entier, Réel, Caractère, Logique)
- utilisez au mieux les structures alternatives pour obtenir une logique élégante et lisible

<sup>46</sup> préparation à l'interrogation de novembre 2011

g) LES STRUCTURES RÉPÉTITIVES (BOUCLES)

Pas d'automatisation sans possibilité de répétition !

Pouvoir exécuter une même séquence d'instructions à de multiples reprises est une fonctionnalité dont on ressent très vite la nécessité (il suffit de se rappeler comment on a essayé d'atteindre le mur en RobotProg).

Encore faut-il pouvoir identifier clairement

- ce qu'il faut répéter,
- pourquoi il faut le répéter
- et surtout maîtriser cette répétition (essentiellement : comment l'arrêter !).

Il nous semble que la première partie de ce texte a répondu (du moins partiellement) à ces questions d'une manière assez simple : tout cela, c'est pour atteindre un but (donc rendre vrai un état !). Pour ce faire, il faut une boucle contenant au minimum une instruction d'action contribuant à rendre vrai cet état (*tantQue non murEnFace : Avancer : finTQ*)

Il nous reste à présent à détailler cela quelque peu ... l'algorithmique (et la programmation) proposent dans ce domaine différents outils dont il faut déterminer ceux qui sont fondamentaux et ceux qui auraient un caractère plus ... 'sucré' ...☺

➤ TantQue ...

```

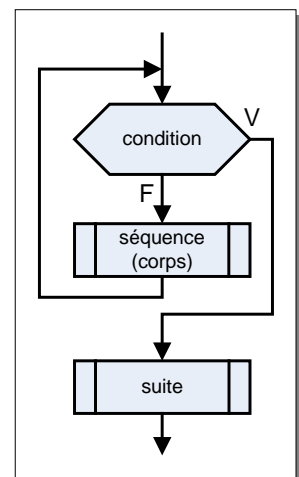
tantQue [non] condition Faire
    séquence
finTQ
  
```

Nous l'avons rencontrée tant et plus dans la première partie du texte : la **boucle avec test antérieur** (avant d'exécuter la séquence) ou encore avec 'principe de précaution'

En termes de buts/états, nous avons pris l'habitude d'utiliser la *formulation suivante* :

*tant qu'un but (état) n'est pas atteint (vrai) ...*

*faire ce qu'il faut pour atteindre (rendre) ce but (cet état vrai)*



Conséquence immédiate :

si la condition est vraie avant la boucle, celle-ci ne sera pas exécutée

(c'est le 'principe de précaution' : tester avant d'agir !)

Danger : la séquence à l'intérieur de la boucle **doit** comporter au moins une instruction (une affectation interne ou externe) qui modifie l'état correspondant à la condition (si non, on a une boucle sans fin)

sur l'automate peleur, la boucle ci-dessous n'a aucune chance ! ...

```

tantQue non marmitePleine
    Si panierVide Alors remplirPanier finSi
    ...
finTQ
  
```

... sans la présence dans la boucle de l'instruction `pelerPatate`, puisque c'est elle – et elle seule ! – qui contribue à modifier l'état de la marmite

Conseil : la boucle tantQue est souvent qualifiée de 'boucle avec condition de continuation'; expliquons-nous sur ce point :

- on part du but à atteindre et on exprime la condition d'arrêt correspondante  
p.ex. : <caseMarquée>  
p.ex. : <robotSurUnePrise>
- on nie ensuite cette condition pour exprimer pourquoi la boucle se poursuit :  
p.ex. : tantQue non caseMarquée  
p.ex. : tantQue non robotSurUnePrise

si la condition d'arrêt est composite (comporte plusieurs états, p.ex. sous forme de comparaisons portant sur différentes valeurs), on procède de même

- exprimer la condition d'arrêt,  
p.ex. revenir au point de départ  $\equiv$  (xRobot = xDépart) **ET** (yRobot = yDépart)
- nier cette condition pour obtenir la condition de continuation de la boucle :  
**NON** ((xRobot = xDépart) ET (yRobot = yDépart))
- habituellement, pour conserver la clarté de la condition, on n'applique pas les lois de Morgan;  
on n'écrit donc pas : (xRobot  $\neq$  xDépart) **OU** (yRobot  $\neq$  yDépart)

Exemple : déterminer qu'un nombre non nul est une puissance de 2 (et laquelle)

- ⇒ première étape : comprendre le problème : ne pas confondre multiple de 2 et puissance de 2 (12 est pair, donc multiple de 2, mais ce n'est pas une puissance de 2)

si n est puissance de 2, alors  $n = 2^p$  (2, 4, 8, 16 ... ne pas oublier que  $1 = 2^0$ ); on divise n successivement par 2 et on s'arrête dès que le reste n'est pas nul ( $12 \div 2 = 6$  reste 0;  $6 \div 2 = 3$  reste 0;  $3 \div 2 = 1$  reste 1) ; il suffit alors de vérifier le dividende (si n est puissance de 2, on a fini sur  $2 \div 2 = 1$  reste 0 puis  $1 \div 2 = 0$  reste 1)

- ⇒ deuxième étape : mise en algorithme

Algorithme puissance2 :

Variable nombre, nbre, puissance : Entier

Début

*# initialisation, encodage*

puissance  $\leftarrow$  0 *# initialisation indispensable !!!*

lire(nombre) *# saisie*

Si nombre > 0 Alors *# partir de 1 !!!*

nbre  $\leftarrow$  nombre *# utiliser une variable de travail*

*# calcul : boucle de divisions successives du nombre par 2*

**tantQue** nbre mod 2 = 0 **Faire** *# donc on n'entre pas si impair*

puissance  $\leftarrow$  puissance + 1

nbre  $\leftarrow$  nbre div 2 *# cette affectation modifie la condition*

**finTQ**

*# puissance de 2 uniquement si on a terminé par 2 div 2, c.à.d 1*

Si nbre = 1 Alors

écrire(nombre, " = 2 exposant ", puissance)

Sinon

écrire(nombre, " n'est pas une puissance de 2")

finSi

finSi

Fin.

### Exemple : valider l'encodage d'un nombre par rapport à un intervalle

```

Algorithme valideur1 :
Constante limiteInf = 23, limiteSup = 55  # les limites à respecter
Variable nombre : Entier                  # le nombre à encoder
Début
  # initialisation, première lecture du nombre
  lire(nombre)
  # si ok on n'entre pas dans la boucle
  tantQue non (nombre entre limiteInf et limiteSup) Faire
    lire(nombre) # redemander : cette affectation modifie la condition
  finTQ
Fin.

```

Bien que la boucle tantQue soit qualifiée de primitive (on pourrait s'en contenter), il existe des situations où elle ne correspond pas parfaitement au problème posé, et son usage (forcé) est quelque peu contre-productif

Une illustration (simple) d'une telle situation est la mission robotProg suivante : *le robot est contre le mur, et on lui demande de faire une fois le tour du terrain et de s'arrêter à son point de départ*

si l'on écrit :

```

Algorithme unTour :
Début
  xDépart ← xRobot      # mémoriser coordonnée x position de départ
  yDépart ← yRobot      # mémoriser coordonnée y position de départ
  tantQue non (xRobot = xDépart ET yRobot = yDépart) Faire
    Si murEnFace Alors tournerDroite finSi
    avancer
  finTQ
Fin.

```

on aura la (mauvaise) surprise de constater que le robot ne fait rien ! en effet, quand il aborde la boucle, sa condition d'arrêt est déjà satisfaite (il part de son point d'arrivée)

on doit dès lors le faire 'bouger' (avancer) avant la boucle, mais comme il faut rester prudent (il part peut-être d'un coin), on est obligé d'écrire :

```

Algorithme unTour :
Début
  xDépart ← xRobot      # mémoriser coordonnée x position de départ
  yDépart ← yRobot      # mémoriser coordonnée y position de départ
  Si murEnFace Alors tournerDroite finSi      # quitter ...
  avancer                                     # ... le point de départ !
  tantQue non (xRobot = xDépart ET yRobot = yDépart) faire
    Si murEnFace Alors tournerDroite finSi
    avancer
  finTQ
Fin.

```

A force d'examiner un tel cas (et il s'en présente assez souvent), qui peut se schématiser comme ci-contre :

séquence <b>tantQue</b> [non] condition <b>Faire</b> séquence <b>finTQ</b>
---

on finit par se dire :

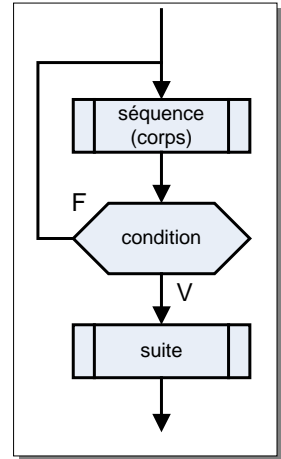
*et si on disposait d'une boucle qui ferait le test après la séquence plutôt qu'avant ???*

➤ **Répéter ... Jusque ...**

**Répéter**  
séquence  
**Jusque** [non] condition

Il s'agit ici d'une **boucle avec test postérieur** (après l'exécution de la séquence) :

En termes de buts/états, nous dirons donc de manière cette fois 'positive' (on exprime cette fois la condition d'arrêt, et non sa négation)



*Répéter*

*faire ce qu'il faut pour atteindre (rendre) un but (un état vrai)*

*Jusqu'à ce que ce but (état) soit atteint (vrai)*

Conséquence immédiate :

même si la condition est vraie avant la boucle,  
le corps de celle-ci sera tout de même exécuté une fois

Danger : la séquence à l'intérieur de l'instruction de boucle doit ici aussi comporter au moins une instruction (une affectation interne ou externe, un appel de procédure) qui modifie l'état correspondant à la condition (sinon, on a une boucle sans fin)

Conseil : la boucle Répéter ... Jusque est souvent qualifiée de 'boucle avec condition d'arrêt'; faut-il vraiment justifier ?

- on part du but à atteindre et on exprime la condition d'arrêt correspondante :  
p.ex. : <caseMarquée>  
p.ex. : <robotSurUnePrise>
- et c'est tout ! cette condition est celle à atteindre après exécution du corps de la boucle  
p.ex. : Répéter Avancer Jusque caseMarquée  
p.ex. : Répéter Avancer Jusque robotSurUnePrise

(cette boucle n'existe pas en RobotProg), mais sur l'automate peleur, si on garantit au départ que la marmite n'est pas pleine, au lieu d'écrire

```

Si panierVide Alors remplirPanier finSi
pelerPatate
tantQue non marmitePleine
  Si panierVide Alors remplirPanier finSi
  pelerPatate
finTQ
  
```

on écrira plutôt (et la séquence grisée ne figure plus qu'une fois)

```

Répéter
  Si panierVide Alors remplirPanier finSi
  pelerPatate
Jusque marmitePleine
  
```



Exemple : valider l'encodage d'un nombre par rapport à un intervalle

```

Algorithme valideur2 :
Constante limiteInf = 23, limiteSup = 55
Variable nombre : Entier
Début
  Répéter
    lire(nombre) # (re)demande : cette affectation modifie la condition
  Jusqu'à nombre entre limiteInf et limiteSup
Fin.

```

---

Exemple : calculer la puissance de 2 d'un nombre positif donné

```

Algorithme puissance2 :
Variable nombre, puissance : Entier
Début
  # initialisation, encodage
  puissance ← 1 # essentiel ! (2 puissance 0)
  lire(nombre)
  Si nombre > 0 Alors
    # calcul : boucle de multiplications successives par 2
    Répéter
      puissance ← puissance * 2
      nombre ← nombre - 1 # cette affectation modifie la condition
    Jusqu'à nombre = 0
    écrire(puissance)
  finSi
Fin.

```

---

Exemple : calculer le temps pour doubler un capital si l'intérêt annuel est fixe

```

Algorithme doublerCapital :
Constante intérêt = 0.045 # 4.5%
Variable durée : Entier,
      capitalDépart, capitalCalculé : Réel
Début
  # initialisations, encodage
  durée ← 0
  lire(capitalDépart)
  Si capitalDépart > 0 Alors
    capitalCalculé ← capitalDépart
    # calcul : boucle de multiplications successives
    Répéter
      durée ← durée + 1 # un an de plus
      capitalCalculé ← capitalCalculé * (1 + intérêt)
      écrire("après ", durée, " ans, capital = ", capitalCalculé)
    Jusqu'à capitalCalculé ≥ 2 * capitalInitial # !!! ≥ 2 et pas = 2 !!!
  finSi
Fin.

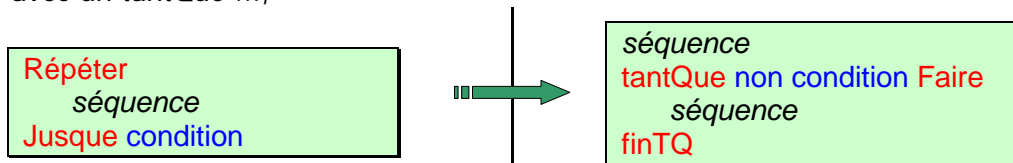
```

---

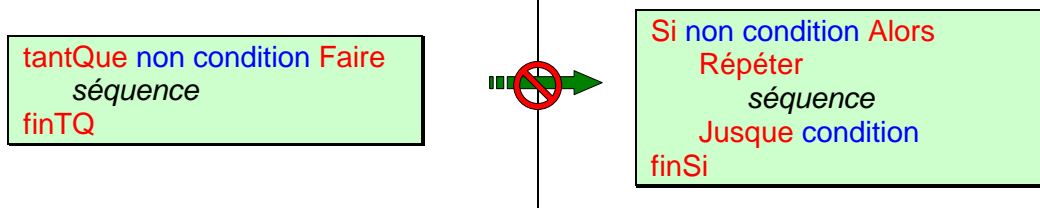
### ➤ Comment choisir ?

Avant d'essayer de répondre à cette question, il faut rappeler que la boucle tantQue peut être considérée comme une 'primitive' (c.-à-d. plus fondamentale) :

- en effet, s'il est très facile (et même élégant ?) d'écrire l'équivalent d'un Répéter ... Jusque ... avec un tantQue ... ,



- le contraire l'est beaucoup moins (élégant)



Pour le reste, la nature du problème et les conditions initiales (le principe de précaution est-il nécessaire ?) déterminent souvent le choix, p.ex. dans l'algorithme doublerCapital de la page précédente, le Répéter ... est 'naturel' ...

... mais d'autres considérations (d'interface homme-machine) interviennent également, par exemple :

- ⇒ on peut écrire une entrée de données validée avec un Répéter ... comme ceci :

```
Algorithme valideur3 :
Constante limiteInf = 23, limiteSup = 55
Variable nombre : Entier
Début
  Répéter
    écrire("introduire un nombre entre ", limiteInf, " et ", limiteSup, " : ")
    lire(nombre)
  Jusque nombre entre limiteInf et limiteSup
Fin.
```

- ⇒ mais si l'on souhaite afficher un message plus explicite en cas d'erreur, il faut utiliser un tantQue :

```
Algorithme valideur4 :
Constante limiteInf = 23, limiteSup = 55
Variable nombre : Entier
Début
  écrire("introduire un nombre entre ", limiteInf, " et ", limiteSup, " : ")
  lire(nombre)
  tantQue non (nombre entre limiteInf et limiteSup) Faire
    écrire("nombre invalide, réessayez : ")
    lire(nombre)
  finTQ
Fin.
```

un dernier mot (pour être complet) : un tantQue n'a pas toujours une condition négative explicite (c'est parfois le Répéter qui exprime la négation) : on se souviendra de robotProg à la recherche d'une porte tout en longeant un mur

tantQue murAGauche	Répéter
avancer	avancer
finTQ	Jusque non murAGauche

### ➤ Exemple récapitulatif

Il s'agit dans une application de proposer des options à l'utilisateur via une interface-texte de type 'menu' (une liste de choix possibles avec pour chacun l'indication du caractère à frapper au clavier)

Il faut valider ce choix et le traiter (ici, simplement écrire un message spécifiant l'option choisie) en boucle; une des options proposées est bien entendu de quitter l'application

On souhaite bien entendu concrétiser par des procédures un algorithme abstrait possédant un état <fini>, du genre :

```
Répéter
  afficherMenu
  validerChoix
  traiterChoix
Jusque fini
```

Algorithme menuValidé :

Variable choix : 0..4, **fini** : Logique

Procédure afficherMenu :

Début

```
  écrire("MENU")
  écrire("====")
  écrire("[1] Ajout")
  écrire("[2] Modification")
  écrire("[3] Suppression")
  écrire("[4] Impression")
  écrire()
  écrire("[0] Quitter")
  écrire()
  écrire("Votre choix : ")
  procChoix
```

Fin;

Procédure validerChoix() :

Début

```
  lire(choix)
  tantQue non (choix entre 0 et 4) Faire
    écrire("choix invalide, réessayez : ")
    lire(choix)
  finTQ
```

Fin;

Procédure traiterChoix :

Début

```
  selonQue choix Vaut
    0 : fini ← vrai
      écrire("au revoir")
    1 : écrire("choix = ajout")
    2 : écrire("choix = modification")
    3 : écrire("choix = suppression")
    4 : écrire("choix = impression")
  finSQ
```

Fin;

Début

```
  Répéter
    afficherMenu
    validerChoix
    traiterChoix
  Jusque fini
```

Fin.

un programme Pascal équivalent est proposé ci-contre :

```
program menuValide;
uses crt;
var choix : 0..4; fini : boolean;

procedure validerChoix;
begin
  readln(choix);
  while not (choix in [0..4]) do begin
    write('choix invalide, réessayez : ');
    readln(choix)
  end
end;

procedure afficherMenu;
begin
  clrscr;
  writeln('MENU');
  writeln('====');
  writeln('[1] Ajout');
  writeln('[2] Modification');
  writeln('[3] Suppression');
  writeln('[4] Impression');
  writeln();
  writeln('[0] Quitter');
  writeln();
  write('Votre choix : ')
end;

procedure traiterChoix;
begin
  case choix of
    0 : begin
        fini := true;
        writeln('au revoir')
      end;
    1 : writeln('choix = ajout');
    2 : writeln('choix = modification');
    3 : writeln('choix = suppression');
    4 : writeln('choix = impression');
  end;
  readkey
end;

begin
  repeat
    afficherMenu;
    validerChoix;
    traiterChoix
  until fini;
end.
```

### ➤ D'autres (types de) boucles ...

Signalons (pour être complet ?)

**Répéter  $n$  Fois**  
*instruction/séquence*  
**finRp**

Elle permet d'exécuter un nombre fixe de fois ( $n$ ) une instruction ou une séquence;  $n$  doit bien entendu être un entier strictement positif, mais peut être indifféremment une constante littérale, une variable, une expression entière, ...

exemple (à la RobotProg ...) : il faut dessiner un carré dont la longueur est choisie par l'utilisateur

```

Algorithme CarréRobot
Variable côté : Entier
Début
  lire(côté)
  Si côté > 0 Alors
    Répéter 4 Fois
      Répéter côté Fois
        Avancer
      finRp
    tournerDroite
  finRp
finSi
Fin

```

**Pour variable de valeurInf à valeurSup Faire**  
*instruction/séquence*  
**finPr**

L'étude de cette boucle particulière dite "avec compteur" sera entreprise dans le seul contexte où elle a du sens (à notre avis) : les algorithmes utilisant la structure de données "Tableaux"

## ➤ Concrètement ...

Au fil du temps, de 'vraies' instructions de boucles sont apparues progressivement dans les langages de programmation (p.ex. elles n'apparaissent que tardivement en Fortran, les premières versions du langage ne connaissant que le IF et le GOTO ...)

### Fortran

### Cobol

Si l'on prend comme référence la première version 'standard' de ce langage (ANS74), elle offre un "vrai" tantQue ... qui s'écrit cependant sous la forme d'un "vrai" Répéter ... Jusque ...; ③.

Puisque ce langage est procédural par nature, le seul moyen d'exécuter une boucle est d'enfermer le corps de la boucle dans une procédure (appelée paragraphe ...et l'invocation d'une procédure correspond au verbe PERFORM)

*Cobol écrit ...*

```
PERFORM
  BOUCLE
UNTIL MARMITE-PLEINE
```

*... qu'il faut lire ...*

```
UNTIL MARMITE-PLEINE
PERFORM BOUCLE
```

*... et en fait, c'est ...*

```
tantQue NON MARMITE-PLEINE Faire
  BOUCLE
finTQ
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.                                PELEUR.
*
DATA DIVISION.
WORKING-STORAGE SECTION.
01 CONTENANCE-SEAU                        PIC 99 VALUE 17.
01 CONTENU-SEAU                          PIC 99.
   88 SEAU-VIDE                          VALUE 0.
01 CONTENU-MARMITE                        PIC 99.
   88 MARMITE-PLEINE                      VALUE 43.
*
PROCEDURE DIVISION.
MAIN SECTION.
  PERFORM INITIALISER.
  PERFORM BOUCLE UNTIL MARMITE-PLEINE.
  DISPLAY "marmite pleine : fini !".
  EXIT PROGRAM.
  STOP RUN.
*
INITIALISER.
  DISPLAY "contenu seau au depart ? ".
  ACCEPT CONTENU-SEAU.
  DISPLAY " contenu marmite au depart ?".
  ACCEPT CONTENU-MARMITE.
*
BOUCLE.
  IF SEAU-VIDE PERFORM REMPLIR-SEAU.
  PERFORM PELER-PATATE.
*
REEMPLIR-SEAU.
  MOVE CONTENANCE-SEAU TO CONTENU-SEAU.
  DISPLAY "seau vide >>> seau rempli !".
*
PELER-PATATE.
  ADD 1 TO CONTENU-MARMITE.
  SUBTRACT 1 FROM CONTENU-SEAU.
  DISPLAY "seau=" CONTENU-SEAU " - marmite=" CONTENU-MARMITE.
*
*****
```

Le seul moyen d'écrire l'équivalent d'un Répéter ... Jusque et d'invoquer une première fois le corps de la boucle ... (p.ex. si on a la garantie que la marmite n'est pas pleine au départ) :

```
PERFORM BOUCLE.
PERFORM BOUCLE UNTIL MARMITE-PLEINE.
```

## Pascal

C'est ici que les boucles telles que nous les connaissons sont réellement introduites : le `while` est un vrai `tantQue` ! Ce langage constitue d'une certaine manière une exception : bien que reconnaissant que – pour toutes les raisons invoquées dans ce chapitre – que le `tantQue` est une 'primitive', grâce à laquelle on peut facilement tout exprimer, N. Wirth ajoute le `repeat ... until`, c-à-d un vrai Répéter ... Jusque ... afin de bien distinguer les boucles à 'test antérieur' des boucles à 'test postérieur'

```

program peleurDePatates;  { * identification du programme/algorithmme * }
uses crt; { * librairie de fonctions/procédures clavier & écran * }
{ * == bloc déclaratif de constantes ===== * }
const contenanceSeau    = 7;
      contenanceMarmite = 43;
{ * == bloc déclaratif de variables ===== * }
var contenuSeau      : integer;
    contenuMarmite   : integer;
{ * == déclaration et définition des fonctions et procédures ===== * }
procedure initialiser; { * fixer contenu initial seau et marmite * }
begin
    contenuSeau := 5;           { * affectation * }
    contenuMarmite := 15;       { * affectation * }
end;
{ * * }
procedure remplirSeau; { * aller au tas remplir le seau * }
begin
    write('seau vide !');      { * état 'avant' * }
    contenuSeau := contenanceSeau; { * affectation 'interne' * }
    writeln(' - seau rempli !') { * état 'après' * }
end;
{ * * }
procedure pelerPatate; { * prendre patate dans seau et mettre dans marmite * }
begin
    contenuSeau := contenuSeau - 1; { * affectation change état du seau * }
    contenuMarmite := contenuMarmite + 1; { * idem change état de la marmite * }
    writeln('seau : ', contenuSeau, ' - marmite : ', contenuMarmite)
end;

{ * ==séquence principale : (version sans variables logiques) ===== * }

```

si l'état de la marmite est inconnu au départ (peut-être pleine ?), on écrira donc

```

begin
    initialiser; { * fixer états initiaux : seau & marmite * }
    while not contenuMarmite = contenanceMarmite do begin { * condition arrêt * }
        if contenuSeau = 0 then remplirSeau; { * modifie état du seau * }
        pelerPatate { * modifie état marmite et seau * }
    end;
    writeln('marmite pleine !'); { * état final de la marmite * }
    readkey
end.

```

mais si on a la garantie que la marmite n'est pas pleine au départ, on écrira plutôt

```

begin
    initialiser; { * fixer états initiaux : seau & marmite * }
    repeat
        if contenuSeau = 0 then remplirSeau; { * modifie état du seau * }
        pelerPatate { * modifie état marmite et seau * }
    until contenuMarmite = contenanceMarmite; { * condition arrêt * }
    writeln('marmite pleine !'); { * état final de la marmite * }
    readkey
end.

{ * * }

```

C

On peut considérer que le C et ses héritiers directs sont en décalage par rapport au Pascal; en effet, le `while` est la seule boucle possible (c'est évidemment un progrès par rapport à ce qu'il y avait avant, puisqu'on a affaire à la boucle primitive) ...

...mais il est mis 'à toutes les sauces' (en C toutes les boucles sont des variations sur le `while`); ainsi par exemple, une boucle avec test postérieur s'écrit : Faire ... tantQue non condition

```
#include <stdio.h>          // librairies de fonctions d'entrée-sortie
// constantes symboliques
#define CONTENANCE_SEAU 17
#define CONTENANCE_MARMITE 43
// prototype des procédures
void initialiser();
void remplirSeau();
void pelerPatate();
// variables (globales)
int contenuSeau, contenuMarmite;
// *** fonction principale *****
int main() {
    initialiser();
    while (! (contenuMarmite == CONTENANCE_MARMITE)) {    // tant que non ...
        if (contenuSeau == 0) remplirSeau();
        pelerPatate();
    }
    printf("\nmarmite pleine : fini !");
    return 0;
}
// *****
// définition des procédures
void initialiser() {
    printf("contenu marmite au depart ? "); // invite
    scanf("%d", &contenuMarmite);           // affectation 'externe'
    printf("contenu seau au depart ? ");     // invite
    scanf("%d", &contenuSeau);               // affectation 'externe'
}
// -----
void pelerPatate() {
    --contenuSeau;           // décrémenter seau
    ++contenuMarmite;        // incrémenter marmite
    printf("seau=%d : marmite=%d \n", contenuSeau, contenuMarmite);
}
// -----
void remplirSeau() {
    contenuSeau = CONTENANCE_SEAU;           // affectation interne
    printf("seau vide ! >>> seau rempli ! \n\n");
}
// *****
```

la version ci-dessus (`main()` est la séquence principale) utilise un `while` avec test antérieur (c-à-d un vrai `tantQue`

la version ci-dessous utilise un `do ... while` avec test postérieur (donc un Répéter ... Jusque ..., sauf qu'il s'écrit Répéter ... tantQue non ...)

```
// *** fonction principale *****
int main() {
    initialiser();
    do {                                                    // répéter
        if (contenuSeau == 0) remplirSeau();
        pelerPatate();
    } while (! (contenuMarmite == CONTENANCE_MARMITE)); // tant que non ...
    printf("\nmarmite pleine : fini !");
    return 0;
}
```

h) EXERCICES

1. écrire un algorithme qui demande un nombre entier compris entre 1 et 3 jusqu'à satisfaction
2. écrire un algorithme qui demande un nombre entier compris entre 10 et 30 jusqu'à satisfaction ; lors de la validation, afficher selon le cas le message d'erreur 'trop petit' ou 'trop grand'
3. étant donné un nombre entier  $n$  introduit au clavier, afficher les  $n$  nombres entiers suivants (p.ex., si on introduit 4, on affiche 5, 6, 7 et 8)
4. étant donné un nombre entier  $n$  introduit au clavier, afficher la somme des entiers jusqu'à ce nombre inclus (p.ex. si on introduit 6, on calcule  $1+2+3+4+5+6$ )
5. écrire un algorithme qui demande à l'utilisateur d'introduire des nombres entre 1 et 100; au fur et à mesure de la saisie, il faut totaliser les nombres introduits; la saisie s'arrête par l'introduction d'un 0, ce qui entraîne l'affichage du total ...
6. rédiger un algorithme procédural d'une machine à voter
  - on fait voter les électeurs entre le candidat A et le candidat B en leur demandant d'introduire la lettre correspondante (au clavier)
  - le but est d'afficher les pourcentages de voix et le vainqueur ;
  - après chaque vote valide, on demande s'il reste des électeurs (on encode le caractère 'O' ou 'N' ; l'introduction de 'N' provoque l'affichage des résultats)
  - l'algorithme doit être fiable à 100% et ne doit pas autoriser d'erreur d'encodage ;
7. soient les règles électorales suivantes :
  - le candidat ayant obtenu au moins 50% des suffrages est élu dès le premier tour
  - en cas de second tour, seuls les candidats ayant obtenu au moins 12.5% des voix au premier tour peuvent se représenter

écrire un algorithme qui permette la saisie (valide) des scores (en % de voix) de quatre candidats. Pour le premier candidat (uniquement), il faut indiquer s'il est élu, battu, en ballottage favorable (participe au second tour en étant en tête) ou en ballottage défavorable (participe au second tour sans être en tête)
8. écrire un algorithme qui demande un nombre entier (base 10) et l'affiche en binaire (p.ex.  $17 \Rightarrow 10001$ )
9. écrire un algorithme qui demande un nombre entier binaire et l'affiche en base 10 (p.ex.  $10001 \Rightarrow 17$ )
10. un nombre (entier) est divisible par 9 si la somme des chiffres qui le constituent est divisible par 9 ; écrire un algorithme qui détermine si un nombre (entier) est ou non divisible par 9

*dans tous ces exercices, n'oubliez pas l'écriture procédurale ...*



11. Lire des données – à valider ! - concernant une série de personnes :

- nom et prénom (type Texte)
- année de naissance (de 1901 à l'année en cours)
- sexe (M/F)

à partir de ces données introduites au clavier (arrêt de la saisie avec introduction - comme nom - de la chaîne "\*\*\*\*" par exemple), le programme doit fournir :

- le nombre de personnes de sexe masculin et leur moyenne d'âge
- le nombre de personnes de sexe féminin et leur moyenne d'âge
- le nombre de personnes de sexe féminin âgées de plus de 30 ans
- le nombre de personnes de sexe masculin nées une année bissextile

12. Calculer la moyenne des cotes d'un ensemble d'étudiants pour un cours donné :

- on demande d'abord combien il y a d'étudiants
- on encode ensuite le nom et la cote de chacun
- l'encodage terminé, on indique :
  - la cote moyenne de l'ensemble
  - le nom de l'étudiant ayant obtenu la meilleure cote
  - le nombre d'étudiants ayant obtenu la moyenne ou plus
  - le pourcentage du nombre d'étudiants n'ayant pas obtenu la moyenne



## 4. OUTILS DE GÉNÉRALISATION : PROCÉDURES ET FONCTIONS

*... la procédure est un des rares outils fondamentaux de l'art de la programmation dont la maîtrise influe de manière décisive sur le style et la qualité du travail du programmeur. La procédure sert à abréger le texte (du programme), et, de façon plus significative, à partager et structurer un programme en composants fermés et logiquement cohérents. Ce cloisonnement est essentiel à la compréhension d'un programme, en particulier si celui-ci est si complexe que son texte prend une longueur impossible à embrasser d'un seul coup d'œil.*

*La structuration en sous-programmes est indispensable, à la fois à la documentation et à la vérification du programme. Par conséquent, il est souvent souhaitable de faire d'une suite d'énoncés (d'une séquence) une procédure, même si elle n'apparaît qu'une seule fois et qu'il n'y a donc aucune possibilité de raccourcir le texte par ce moyen.*

*Des informations sur les variables (celles dont la valeur est utilisée ou modifiée par la procédure) ou sur les conditions que les arguments doivent satisfaire, peuvent apparaître de façon commode dans l'en-tête de la procédure.*

*... nous appellerons procédures des séquences d'instructions portant un nom; si ces procédures représentent également une valeur résultante et peuvent servir dans des expressions, on les appellera des fonctions.*

*... il arrive fréquemment que certaines variables soient utilisées dans une suite d'énoncés (une séquence) sans avoir la moindre signification en-dehors des ces énoncés. La clarté du programme est nettement améliorée si l'on peut faire apparaître clairement la région où ces variables sont significatives.*

*La procédure (ou la fonction) apparaît comme l'unité textuelle naturelle pour délimiter le domaine de validité des ces variables locales.*

*... les cas est tout aussi fréquent où une certaine séquence apparaît à plusieurs endroits sous une forme pas tout à fait identique mais très semblable. ... dans ce cas, les séquences à abréger peuvent être transformés en un schéma de procédure (de fonction) abstrait.*

*Les entités qu'il faut encore préciser dans chaque utilisation sont les paramètres des procédures (et des fonctions).*

*N. Wirth<sup>47</sup>, Introduction à la programmation systématique, Masson, 1981*

Nous avons déjà à plusieurs reprises eu l'occasion de rencontrer cette notion fondamentale de procédure (dans la première et la seconde partie de ce texte). Tout en effet encourage le programmeur à en faire un large usage : c'est l'expression naturelle d'une pensée structurée, 'descendante', qui va du général au particulier (de l'algorithme abstrait à sa concrétisation en définissant au passage les données nécessaires, constantes et variables, et en utilisant la seule action élémentaire disponible : l'affectation ... )

Fonctions et/ou procédures sont des outils permettant de structurer les algorithmes. Ils sont présents dans la plupart des langages de programmation. Fonctions ou procédures succèdent ainsi à la notion de "sous-programme", notion introduite dans les premiers langages de programmation. L'objectif est de permettre l'écriture d'un algorithme par juxtaposition de "morceaux" résolvant chacun un sous-problème particulier.

Il faut à présent pousser la réflexion plus loin et

- d'une part établir d'une part la distinction nette entre procédure (action) et fonction (valeur)
- d'autre part envisager les procédures et les fonctions comme des outils réutilisables et offrant un haut niveau d'abstraction (formalisation et passage de paramètres)

<sup>47</sup> un des pères de l'expression algorithmique moderne (pseudo-code) et créateur du langage Pascal (entre autres ...)

## 4.1. PROCÉDURES

### a) GÉNÉRALITÉS

Rappels : on a défini précédemment la procédure comme une généralisation du concept d'action, à partir des considérations suivantes :

- Une instruction est l'outil permettant de spécifier une action à exécuter
- Une séquence est une suite ordonnée d'instructions qui seront exécutées dans l'ordre strict où elles ont été rédigées.
- Une procédure est une séquence d'instructions à laquelle on donne un nom et qui pourra être utilisée de la même manière que les actions élémentaires.

La procédure est donc un outil de base de structuration (de la solution) d'un problème, permettant le 'découpage' en sous-problèmes : on s'efforcera de rendre ces procédures aussi indépendantes que possible (chacune ne 'voit' que ce qui la concerne et rien d'autre)

Exemple 1 (minimaliste mais représentatif) : calculer le résultat d'une division réelle : concrétisation d'un algorithme abstrait à l'aide de variables, découpage du problème en procédure spécialisées et indépendantes les unes des autres, mais communiquant via les variables globales

```
Algorithme Division : # phase 1 : algo abstrait
Début
    saisirDonnées
    traiterDonnées
    afficherRésultats
Fin.
```

---

```
Algorithme Division : # phase 2 : concrétisation : déclaration de données
Variable nombre1, nombre2, résultat : Réel # données nécessaires
Début
    saisirDonnées
    traiterDonnées
    afficherRésultats
Fin.
```

---

```
Algorithme Division : # phase 3 : concrétisation : procédures
Variable nombre1, nombre2, résultat : Réel # variables globales

Procédure saisirDonnées : # saisie des données
Début # saisie indépendante de ce que l'on fera des données ensuite
    écrire("introduire nombre 1 : ")
    lire(nombre1) # accès à la variable globale
    écrire("introduire nombre 2 : ")
    lire(nombre2) # accès à la variable globale
Fin

Procédure traiterDonnées : # traitement des données, ...
Début # indépendant de la manière dont les données ont été saisies
    Si nombre2 ≠ 0 Alors # en fonction des données d'entrée
        résultat ← nombre1 / nombre2 # calcul d'un résultat (variable globale)
    finSi
Fin

Procédure afficherRésultat : # affichage résultat, ...
Début # devrait être indépendante des données d'entrée !
    Si nombre2 ≠ 0 Alors # mais il y a une dépendance ici !
        écrire(résultat) # idéalement, elle ne devrait faire que ceci
    Sinon
        écrire("calcul impossible")
    finSi
Fin
```

```

Début # séquence principale : idéalement reste un algorithme abstrait (ne manipule pas directement les données mais agit par délégation via les procédures)
    saisirDonnées
    traiterDonnées
    afficherRésultats
Fin.

```

---

## b) PLUS D'INDÉPENDANCE : LES ÉTATS

Pour augmenter l'indépendance de chacune des procédures, on peut avoir recours – par exemple – à des variables logiques, implémentation concrète des états binaires.

Pour éviter qu'une procédure ait accès à des données qui ne la concernent pas, on va demander à une autre procédure – parce qu'elle est la mieux placée pour le faire, - de déterminer la valeur d'un état et de mettre celui-ci - via une variable logique - à la disposition des autres procédures

```

Algorithme Division: # phase 4 : concrétisation : indépendance
Variable nombre1, nombre2, résultat : Réel
    ok : Logique # pour assurer une meilleure indépendance

Procédure saisirDonnées : # saisie des données
Début # indépendant de ce que l'on fera des données ensuite
    écrire("introduire nombre 1 : ")
    lire(nombre1)
    écrire("introduire nombre 2 : ")
    lire(nombre2)
Fin;

Procédure traiterDonnées : # traitement des données
Début # bien placé pour déterminer si le traitement est possible
    ok ← (nombre2 ≠ 0) # état : le mettre ici ! et pas dans la saisie !
    Si ok Alors
        résultat ← nombre1 / nombre2
    finSi
Fin;

Procédure afficherRésultat : # affichage résultat, indépendant des données
Début
    Si ok Alors # ceci est devenu indépendant
        écrire(résultat)
    Sinon
        écrire("calcul impossible")
    finSi
Fin;

Début # séquence principale : reste l'algorithme abstrait
    saisirDonnées
    traiterDonnées
    afficherRésultats
Fin.

```

---

**Exemple 2** (encore plus minimaliste mais tout aussi représentatif) : échanger le contenu de deux variables de même type (outil indispensable pour envisager le tri procédural, cfr. chapitre ultérieur sur les tableaux)

```

Algorithme Échanger : # concrétisation : déclaration de données
Variable donnéé1, donnée2, échange : Entier # données nécessaires

Procédure saisirDonnées : # saisie des données
Début # indépendant de ce que l'on fera des données ensuite
    écrire("introduire donnée 1 (nbre entier) : ")
    lire(nombre1)
    écrire("introduire donnée 2 (nbre entier) : ")
    lire(nombre2)
Fin;

Procédure échangerDonnées : # échange des données
Début # passer par une troisième variable
    échange ← donnéé1
    donnéé1 ← donnée2
    donnée2 ← échange
Fin

Procédure afficherDonnées : # affichage des données
Début # indépendant de la manière dont elles ont été produites
    écrire("donnée 1 : ", donnéé1)
    écrire("donnée 2 : ", donnée2)
Fin

Début
    saisirDonnées
    écrire("données avant échange : ")
    afficherDonnées
    échangerDonnées
    écrire("données après échange : ")
    afficherDonnées
Fin.

```

mais en y regardant de plus près, on peut s'interroger sur la variable échange; son rôle est clair : pour échanger le contenu de deux variables, il en faut une troisième ! dans la mesure où cet échange s'effectue dans une procédure, seule cette procédure a besoin de cette variable (la mettre à la disposition de l'algorithme lui-même et des autres procédures n'a pas de sens); aussi écrira-t-on désormais :

```

Algorithme Échanger : # concrétisation : déclaration de données
Variable donnéé1, donnée2 : Entier # données nécessaires à l'algo
...

Procédure échangerDonnées : # échange des données
Variable échange : Entier # variable nécessaire à la procédure et à elle seule
Début # passer par une troisième variable
    échange ← donnéé1
    donnéé1 ← donnée2
    donnée2 ← échange
Fin

...

```

La distinction entre des variables déclarées dans l'algorithme et d'autres déclarées dans les procédures est donc un outil important de structure (au niveau des données cette fois) et d'indépendance (au niveau des données et des procédures, encore elles !)

### c) LA PORTÉE DES VARIABLES ET DES PROCÉDURES

Les données (variables) qui ont été déclarées au niveau du bloc déclaratif de l'algorithme sont qualifiées de variables globales; celles qui sont déclarées au niveau d'une procédure sont qualifiées de variables locales (à cette procédure).

Définissons un concept nouveau :

*On désigne par **portée** d'un objet (constante, variable, procédure) le domaine de visibilité de cet objet. C'est donc l'ensemble des endroits où l'objet est connu.*

Les variables globales ont une portée globale car elles peuvent être vues (et modifiées) aussi bien dans la séquence principale que dans chacune des procédures;

Les variables locales ne sont accessibles qu'à la seule procédure qui les a déclarées.

Habituellement, les constantes (parce que leur valeur ne peut pas changer) ont un statut implicite de données globales; elles sont donc généralement déclarées (globalement) en tête de l'algorithme.

Quant aux procédures, elles ont également une portée globale<sup>48</sup> (elles sont évidemment accessibles depuis la séquence principale de l'algorithme), mais une procédure peut invoquer une autre procédure, à condition de respecter la règle suivante : *une procédure (disons P1) ne peut en invoquer une autre (disons P2) que dans la mesure où celle-ci (P2) a été déclarée (définie) au préalable (avant P1)*

La notion de portée de variable est indissociable de la question suivante (posée via l'exemple ci-dessous) : que se passe-t-il quand une variable globale et une variable locale portent le même nom ?

```

Algorithme algoTest :
  Variable x, y, z : Entier    # variable globale

  ...

  Procédure procTest : # procédure
  Variable x : Entier    # variable locale à la procédure
  Début
    ...
    x ← 127    # affectation de la variable locale
    ...
  Fin

  ...

  Début    # séquence principale
  ...
  procTest    # appel de la procédure
  écrire(x)    # erreur probable !
  ..x ← 255    # affectation de la variable globale
  ...
  Fin
  
```

La réponse est simple :

- la séquence de l'algorithme n'a accès qu'aux seules variables globales (donc sa tentative d'écriture concerne la variable globale qui à ce stade est non initialisée !)
- la variable x déclarée dans la procédure est locale à cette procédure et 'masque' la variable globale de même nom (qui lui est donc devenue inaccessible); à la sortie de la procédure, la variable locale x n'existe tout simplement plus !

<sup>48</sup> on peut imaginer qu'une procédure P2 soit déclarée et définie au sein d'une autre procédure P1; P2 est alors locale à P1 (seule celle-ci peut l'invoquer); cette fonctionnalité est disponible en langage Pascal; cependant nous ne l'évoquerons pas dans ce cours d'initiation

## 4.2. FONCTIONS

### a) GÉNÉRALITÉS : RAPPEL SUR LA NOTION DE VALEUR

Dernier jalon très important dans la découverte des outils de l'algorithmique impérative procédurale : la fonction.

De la même façon que la procédure est une généralisation du concept d'action :

*partout où l'on peut mettre une séquence d'une (ou plusieurs) action(s), on peut la remplacer par un appel de procédure*

la fonction est une généralisation du concept de valeur :

*partout où l'on peut placer une valeur, on peut placer un appel de fonction*

Commençons par rappeler la notion de valeur :

- jusqu'à présent, les valeurs se présentaient sous trois formes :

- la valeur littérale,
- la constante symbolique
- et la (valeur, le contenu d'une) variable

```
Constante c = 22
Variable v1, v2 : Entier
fini : Logique
```

- où peut-on mettre une telle valeur ?

- à droite d'un symbole d'affectation (à gauche : c'est toujours obligatoirement une variable) :

```
v2 ← 13      # valeur littérale
v2 ← c       # valeur symbolique : constante
v2 ← v1      # attention : mettre dans v2 la valeur (le contenu) de v1
```

- comme terme ou facteur dans une expression :

```
v2 ← v2 + c * v1 + 10    # expression numérique
```

- dans une expression de comparaison (condition) : à gauche comme à droite de l'opérateur relationnel

```
Si v1 = v2 Alors ...    # si valeur de v1 = valeur de v2
fini ← (v1 - v2 < c)    # valeur d'expression de comparaison affectée à une
                        # variable logique
tantQue non fini ...    # valeur de la variable logique
```

La quatrième forme de valeur est la fonction (plus rigoureusement, la valeur retournée par l'appel d'une fonction, on va préciser ...)

Des fonctions, nous en avons déjà rencontré (elles ont accompagné notre découverte des types de données scalaires, elles offrent à côté des opérateurs, des possibilités diverses, il suffit de penser aux fonctions mathématiques sur les types numériques ...), et nous les avons utilisées 'intuitivement', comme une valeur, sans trop nous poser de question quant à leur nature exacte :

```
Variable réponse : caractère
...
Répéter
  lire(réponse)
  réponse ← minusc(réponse)           # lowercase() en Pascal
Jusque réponse parmi ('o', 'n')
```



```

Variable prix : Entier
...
lire(prix)
prix ← valeurAbsolue(prix)                                # abs() en Pascal

Variable a, b, c, delta, x1, x2 : Réel
lire(a); lire(b); lire(c)
Si a ≠ 0 Alors
    delta ← b * b - 4 a * c
    Si delta ≥ 0 Alors
        x1 ← (- b + racineCarrée(delta)) / 2 * a          # sqrt() en Pascal
        x2 ← (- b - racineCarrée(delta)) / 2 * a
    finSi
finSi

Variable c : Caractère
c ← 'H'
c ← successeur(c)                                         # succ() en Pascal

```

## b) LA FONCTION : GÉNÉRALISATION DE LA NOTION DE VALEUR

Donc, si on peut placer une fonction à tout endroit où l'on peut placer une valeur, c'est que la fonction est d'une certaine manière un 'outil procédural' qui calcule et renvoie une valeur !

Partons d'une image mathématique de la fonction (elle nous est peut-être plus familière) :

quand on définit les deux fonctions  $f_1(x)$  et  $f_2(x)$  comme suit :

$$f_1(x) = 3.x^2 + 2.x - 3$$

$$f_2(x) = 1 + \tan(x),$$

- on donne un nom à chaque fonction :  $f_1$  et  $f_2$
- on rédige (à droite du symbole =) une expression algébrique qui est formellement *'ce que doit faire et comment'* la fonction (une forme rudimentaire d'algorithme)
- on associe une lettre  $x$  au nom de la fonction (entre parenthèses) et on utilise cette lettre  $x$  dans son expression (son 'algorithme') pour symboliser n'importe quelle valeur du domaine de définition des fonctions (p.ex. les réels) ... *étant donné un réel  $x$ , calculer et fournir la valeur  $3x^2 + 2x - 3$*
- enfin, quand on en a besoin, on 'appelle' (invoque) la fonction en utilisant son nom en lui communiquant une valeur précise de  $x$  et elle 'répond' par la valeur de son expression pour cette valeur de  $x$  : une fonction ne renvoie jamais qu'une seule valeur !
- ainsi :  $f_1(2)$  vaut 13,  $f_1(0)$  vaut -3,  $f_2(0)$  vaut 1 et  $f_2(\pi/4)$  vaut 2

En algorithmique (et en programmation), on procède de même, mais avec des concepts un peu moins 'matheux' et plus adaptés au domaine de l'algorithmique (on va pouvoir écrire des fonctions qui n'ont aucun caractère mathématique) ... en attendant, on écrira la définition de la première fonction 'mathématique' ci-dessus comme ceci (détails plus loin) :

```

Fonction f1(x : Réel) : Réel          # attention au typage ! cfr. plus loin
Variable valTmp : Réel                 # variable locale de travail (facultative)
Début
    valTmp ← 3 * x * x + 2 * x - 3      # expression à évaluer et affectation de son résultat
    renvoie(valTmp)                   # renvoi de la valeur au 'monde extérieur'
Fin

```

et on l'utilisera comme ceci (c'est-à-dire comme ce qu'elle est : une valeur!)

```

Algorithme fctTest
Variable v1, v2, v3 : Réel
...
Début
...
  v1 ← f1(2)           # invoque f1 en lui transmettant 2; valeur retour affectée à v1
  v2 ← f1(v1) + 5
  Si f1(2) = 3 Alors ...# invoque f1 en lui transmettant 2; valeur retour comparée à 3
...
Fin

```

Avant d'aller plus loin, retour en arrière, une (dernière ?) fois sur l'"algorithme du peleur" afin de découvrir la réelle importance des fonctions (importance au moins égale – sinon plus - à celle des procédures)

Partant de la version abstraite (purement fonctionnelle)

```

Algorithme Peleur :
Début
  TantQue non marmitePleine Faire
    Si seauVide Alors remplirSeau FinSi
    pelerPatate
  FinTQ
Fin.

```

nous avons rédigé une première version procédurale concrète comme ceci :

```

Algorithme Peleur :

# Déclaration des données

Constante contenanceSeau = 15, contenanceMarmite = 33
Variable contenuSeau, contenuMarmite : Entier

# Déclaration/Définition des procédures

Procédure initialiser :
Début
  lire(contenuSeau)      # combien de patates dans le seau au début
  lire(contenuMarmite)   # combien de patates dans la marmite au début
Fin

Procédure remplirSeau :
Début
  contenuSeau ← contenanceSeau # mettre toutes les patates possibles
Fin

Procédure pelerPatate :
Début
  contenuSeau ← contenuSeau - 1      # une patate de moins
  contenuMarmite ← contenuMarmite + 1 # une patate de plus
Fin

# Algorithme proprement dit : séquence principale

Début
  initialiser
  TantQue non contenuMarmite = contenanceMarmite Faire
    Si contenuSeau = 0 Alors remplirSeau FinSi
    pelerPatate
  finTQ
Fin.

```

puis, considérant que l'algorithme principal avait perdu son caractère abstrait, (en effet, on y fait référence explicitement aux données, littérales, constantes et/ou variables), nous avons corrigé le tir en introduisant les variables logiques comme concrétisation des états (marmite-Pleine, seauVide) comme ceci :

**Algorithme** Peleur :

```

Constante contenanceSeau = 15, contenanceMarmite = 33
Variable contenuSeau, contenuMarmite : Entier
           seauVide, marmitePleine : Logique           # deux états

Procédure initialiser :
Début
    lire(contenuSeau)           # combien de patates dans le seau au début
    lire(contenuMarmite)        # combien de patates dans la marmite au début
    marmitePleine ← (contenuMarmite = contenanceMarmite)    # état
    seauVide ← (contenuSeau = 0)                             # état
Fin

Procédure pelerPatate :
Début
    contenuSeau ← contenuSeau - 1
    contenuMarmite ← contenuMarmite + 1
    marmitePleine ← (contenuMarmite = contenanceMarmite)    # état
    seauVide ← (contenuSeau = 0)                             # état
Fin

...# autres procédures

Début
    initialiser
    TantQue non marmitePleine Faire
        Si seauVide Alors remplirSeau FinSi
        pelerPatate
    finTQ
Fin.
```

mission accomplie du point de vue de la séquence principale, mais à quel prix ! nous avons à présent chargé deux procédures d'une responsabilité qui n'est pas vraiment la leur ! (p.ex. le 'métier' de la procédure pelerPatate et de prendre une patate, la peler et la déposer, pas de s'interroger sur le contenu du seau et de la marmite et d'en informer les autres !)

comme on y trouve deux fois le même couple d'instructions, réutilisation oblige, on aurait pu passer par une procédure supplémentaire

```

Procédure publierEtats : # informer tous azimuth ...
Début
    marmitePleine ← (contenuMarmite = contenanceMarmite)    # état
    seauVide ← (contenuSeau = 0)                             # état
Fin

Procédure initialiser :
Début
    lire(contenuSeau)           # combien de patates dans le seau au début
    lire(contenuMarmite)        # combien de patates dans la marmite au début
    publierEtats                # informer ...
Fin

Procédure pelerPatate :
Début
    contenuSeau ← contenuSeau - 1
    contenuMarmite ← contenuMarmite + 1
    publierEtats                # informer ...
Fin
```

mais le problème reste entier ..., cette publication n'est pas le job de ces procédures !

C'est ici que les fonctions vont révéler leur intérêt : récrivons comme ceci :

**Algorithme** Peleur :

Constante    contenanceSeau = 15, contenanceMarmite = 33  
Variable    contenuSeau, contenuMarmite : Entier

**# déclarations/définitions de fonctions**

**Fonction seauVide() : Logique :** # renvoie l'état vide ou non du seau

Début

**renvoie** (contenuSeau = 0)    # comparaison => vrai ou faux

Fin

**Fonction marmitePleine() : Logique :** # renvoie l'état plein ou non de la marmite

Début

**renvoie** (contenuMarmite = contenanceMarmite) # compar. => vrai ou faux

Fin

**# déclarations/définitions de procédures**

Procédure **initialiser** :

Début

    lire(contenuSeau)    # combien de patates dans le seau au début

    lire(contenuMarmite)    # combien de patates dans la marmite au début

Fin

Procédure **remplirSeau** :

Début

    contenuSeau ← contenanceSeau#    # mettre les patates possibles

Fin

Procédure **pelerPatate** :

Début

    contenuSeau ← contenuSeau - 1

    contenuMarmite ← contenuMarmite + 1

Fin

**# algorithme proprement dit : séquence principale**

Début

**initialiser**

**TantQue** non **marmitePleine()** **Faire**    # je suis intéressé par ceci

**Si** **seauVide()** **Alors** **remplirSeau** **FinSi**    # et par ceci

**pelerPatate**

**finTQ**

Fin.

quelle(s) agréable(s) surprise(s) :

- les deux variables logiques (états concrets) ont disparu
- tout comme leur modification dans deux procédures, qui retrouvent leur 'métier' d'origine !

en lieu et place :

- deux fonctions calculant chacune (par comparaison entre variables globales et valeurs appropriées) et renvoyant une valeur vrai/faux (c-à-d. l'équivalent d'un état)
- l'invocation – par celui qui en a besoin et au moment où il en a besoin (ici la séquence principale) – de ces fonctions, considérées à ce moment comme une valeur logique

### 4.3. NOTION DE PARAMÈTRE

Une déclaration de fonction utilise le modèle général suivant :

```

Fonction nomDeLa Fonction ([nom et type des paramètres]) : type de la valeur de retour :
Début
...
renvoie(valeur)
Fin
  
```

- il n'est pas surprenant d'y trouver :

- le mot Fonction (l'équivalent des mots Algorithme et Procédure) accompagné d'un nom (identificateur symbolique) choisi par le programmeur pour pouvoir ensuite invoquer (faire exécuter) la fonction :

```
Fonction min(...) ... :
```

```
Fonction cube(...) ... :
```

```
Fonction estImpair(...) ... :
```

- la déclaration du type de la valeur de retour de la fonction (elle calcule une valeur et toute valeur est nécessairement typée) : on se contentera pour l'instant de fonctions calculant une valeur scalaire (donc de type Entier, Réel, Caractère, Logique, Texte)

```
Fonction min(...) : Réel :
```

```
Fonction cube(...) : Entier :
```

```
Fonction estImpair(...) : Logique :
```

- entre les parenthèses apparaît la déclaration du ou des paramètre(s) d'entrée : c'est par ce canal que sont transmises à la fonction les valeurs qu'elle devra utiliser pour calculer sa valeur de retour; ces paramètres portent bien entendu chacun un nom accompagné de sa déclaration de type; dans le cadre de la déclaration/définition d'une fonction, ils sont qualifiés de paramètres formels; dans une fonction, ils ont toujours un statut de variable locale

```
Fonction min(x1 : Réel, x2 : Réel) : Réel :
```

```
Fonction cube(x : Entier) : Entier :
```

```
Fonction estImpair(n : Entier) : Logique :
```

- enfin, dans le corps de la fonction (en principe la dernière ligne, c'est plus 'propre') figure obligatoirement une instruction dite 'de retour' permettant à la fonction de mettre la valeur calculée à disposition de l'instruction qui l'a invoquée

```
Fonction min(x1 : Réel, x2 : Réel) : Réel : # renvoie le plus petit
```

```
Début
```

```
  Si n1 < n2 Alors Renvoie(n1) Sinon Renvoie(n2) finSi
```

```
Fin
```

```
Fonction cube(x : Entier) : Entier : # renvoie x au cube
```

```
Début
```

```
  Renvoie(x * x * x)
```

```
Fin
```

```
Fonction estImpair(n : Entier) : Logique : # renvoie vrai ou faux
```

```
Variable reste : Entier
```

```
# variable de travail pour l'exemple
```

```
Début
```

```
  reste ← n mod 2
```

```
# calcul du reste de la division par 2
```

```
  renvoie(reste = 1)
```

```
# impair si ce reste est 1
```

```
Fin
```

- au moment d'appeler la fonction, l'appelant (une instruction dans l'algorithme principal, dans une procédure ou même une autre fonction) lui communique les valeurs à utiliser pour calculer la valeur de retour : ces valeurs – qualifiées de paramètres réels cette fois – correspondent à des valeurs littérales, des constantes ou des variables appartenant à l'environnement de données de l'appelant

Constante échelle = 1000.0

Variable volt1, volt2, mesure : Réel

...

**Fonction** min(n1 : Réel, n2 : Réel) : Réel : # renvoie le plus petit  
# soient n1 et n2 des valeurs réelles ...

Début

Si n1 < n2 Alors **Renvoie**(n1) Sinon **Renvoie**(n2) finSi

Fin

...

Début

...

mesure ← min(250.3, 325.5) # valeurs littérales

mesure ← min(échelle, v1) # constante, variable

mesure ← min(v1, v2) + min(échelle, v2) # une expression

Si min(v1, v2) = v1 Alors ... Sinon ... # dans une alternative

tantQue min(v1, v2) - mesure < échelle ... # piloter une boucle

...

Fin

En résumé : une fonction est un outil de structure algorithmique réutilisable qui

- est capable de prendre en charge de manière autonome un traitement consistant à calculer une valeur
- possède un nom
- ne peut en aucun cas modifier l'état de l'algorithme; elle ne peut donc modifier (par affectation interne ou externe) les variables globales
- en lieu et place, possède des paramètres formels nommés et typés qui jouent le rôle de variables locales auxquels sont transmises les valeurs des variables globales (on utilisera le terme de passage de paramètre par copie/valeur)
- peut cependant pour réaliser sa 'mission' utiliser les constantes globales

La ligne de déclaration/définition de la fonction, rassemblant son nom, les noms et types des paramètres d'entrée et le type de la valeur de retour est appelée prototype ou signature de la fonction

Exemple :

Algorithme testFct :

Variable A, B, C, D, E, F : Entier

...

**Fonction** exSomme(x, y : Entier) : Entier # prototype, signature

Début

**renvoie** ( 2 \* (x + y) )

Fin

...

Début

...

lire(A) # saisie

lire(B) # saisie

# invocation

C ← exSomme(A, B)

lire(D)

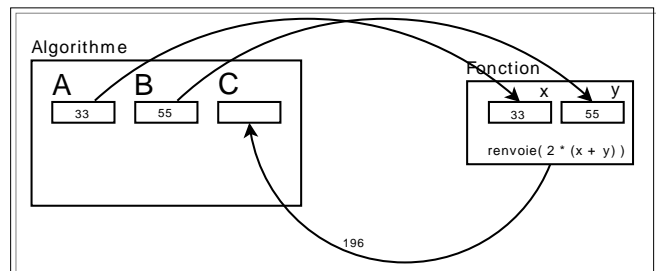
lire(E)

# invocation

F ← exSomme(D, E)

...

Fin



Autre exemple :

```

Algorithme calculPrix
Constante tva = 0.21
Variable prixHtval, prixHtva2 : Réel
...
Fonction prixTvaC(pr : Réel) : Réel           # prototype, signature
Début
    renvoie(pr * (1 + tva))
Fin
...
Début
...
    lire(prixHtval); écrire(prixTvaC(prixHtval))    # invocation directe
    lire(prixHtva2); écrire(prixTvaC(prixHtva2))    # invocation directe
...
Fin

```

Les fonctions sont des outils particulièrement adaptés pour écrire des saisies de données avec validation; par exemple, au lieu d'écrire

```

Algorithme saisieNombres
Constante vMin1 = 10, vMax1 = 100, vMin2 = 1, vMax2 = 20
Variable nombre1, nombre2 : Entier
...
Début
...
    écrire("introduire un nombre entre ", vMin1, " et ", vMax1, " : ")
    lire(nombre1)
    tantQue non (nombre1 entre vMin1 et vMax1) faire
        écrire("valeur hors intervalle, recommencez : ")
        lire(nombre1)
    finTQ
...
    écrire("introduire un nombre entre ", vMin2, " et ", vMax2, " : ")
    lire(nombre2)
    tantQue non (nombre2 entre vMin2 et vMax2) faire
        écrire("valeur hors intervalle, recommencez : ")
        lire(nombre2)
    finTQ
...
Fin

```

on écrira un validateur réutilisable à l'aide d'une fonction correctement paramétrée

```

Algorithme saisieNombres
Constante vMin1 = 10, vMax1 = 100, vMin2 = 1, vMax2 = 20
Variable nombre1, nombre2 : Entier
...
Fonction lireEntier(min, max : Entier) : Entier
Variable n : Entier
Début
    écrire("introduire un nombre entre ", min, " et ", max, " : ")
    lire(n)
    tantQue non (n entre min et max) faire
        écrire("valeur hors intervalle, recommencez : ")
        lire(n)
    finTQ
    renvoie(n)
Fin
...
Début
...
    nombre1 ← lireEntier(vMin1, vMax1)
    nombre2 ← lireEntier(vMin2, vMax2)
...
Fin

```

#### 4.4. PROCÉDURES PARAMÉTRÉES

Revenons à un petit algorithme procédural écrit précédemment : il s'agissait d'échanger le contenu de deux variables, ce que nous avons rédigé comme suit :

```

Algorithme Échanger :
Variable donnéé1, donnée2 : Entier           # données nécessaires à l'algo
...
Procédure échangerDonnées :                  # échange des données
Variable échange : Entier                    # variable nécessaire à la procédure et à elle seule
Début                                       # passer par une troisième variable
    échange ← donnéé1
    donnéé1 ← donnée2
    donnée2 ← échange
Fin
...

```

Que fait la procédure échangerDonnées : elle ne sait échanger que deux variables globales donnéé1 et donnée2. Et s'il fallait dans le même algorithme échanger également deux autres variables donnée3 et donnée4, faudrait-il une autre procédure d'échange ?

```

Algorithme soluProbl :
Variable donnéé1, donnée2, donnée3, donnée4 : Entier           # var. globales
...
Procédure échangerDonnées1 :                      # échange des données
Variable échange : Entier                            # variable locale
Début
    échange ← donnéé1
    donnéé1 ← donnée2
    donnée2 ← échange
Fin

Procédure échangerDonnées2 :
Variable échange : Entier
Début
    échange ← donnée3
    donnée3 ← donnée4
    donnée4 ← échange
Fin
...

```

Non bien sûr : le dernier acte de l'écriture procédurale réutilisable indépendante se joue ici : pour rendre une procédure aussi indépendante que possible des variables globales (sur lesquelles elle est censée travailler), on va également lui transmettre ces données comme paramètres et dès lors on pourra écrire :

```

Algorithme soluProbl :
Variable donnéé1, donnée2, donnée3, donnée4 : Entier           # var. globales
...
Procédure échangerDonnées(ref n1, ref n2 : Entier) :          # échange données
Variable échange : Entier                                       # variable locale à la procédure
Début
    échange ← n1
    n1 ← n2
    n2 ← échange
Fin

Début
...
lire(donnéé1); lire(donnée2)
lire(donnée3); lire(donnée4)
échangerDonnées(donnéé1, donnée2)
échangerDonnées(donnée3, donnée4)
...
Fin

```



Si l'on veut considérer les outils procéduraux que sont les fonctions et les procédures comme des outils réutilisables (c.-à-d. invoqués à plusieurs reprises, avec des données différentes), il faut bien comprendre le rôle essentiel que jouent les paramètres : un canal d'échange de données entre l'appelant et l'appelé.

Ce concept de réutilisation de code avec échange de données peut être étendu aux procédures, mais celles-ci sont des outils d'action (donc modifiant a priori des variables en utilisant l'affectation) : on distinguera alors

- les procédures qui utilisent des données passées par copie/valeur (même principe que les paramètres des fonctions) et qui ne modifient pas les données transmises par l'appelant
- les procédures qui modifient les données transmises par l'appelant; celles-ci leur sont alors transmises par référence/adresse

#### a) ... SANS MODIFICATION DE DONNÉES

une déclaration de procédure avec passage de paramètres par copie/valeur utilise le modèle général suivant :

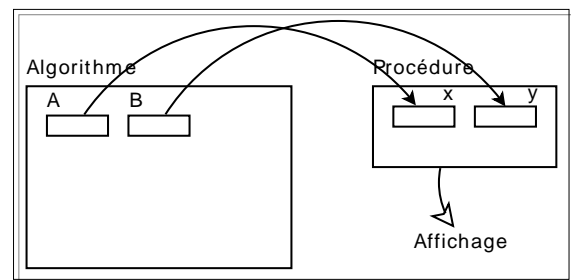
```
Procédure nomDeLaProcédure (nom et type des paramètres) :
Début
...
Fin
```

*# corps de la procédure*

l'exemple typique d'une procédure avec paramètre transmis par copie/valeur est la procédure d'affichage standard `écrire(...)` : en aucun cas les variables qui lui sont confiées ne peuvent être modifiées

comme dans le cas des fonctions, les paramètres ainsi transmis doivent être considérés comme des variables locales à la procédure

```
Algorithme testPrcl1 :
Variable A, B, C, D : Entier
...
Procédure affiche(x, y : Entier) :      # prototype, signature
Début
    écrire( 2 * (x + y) )
Fin;
...
Début
    ...
    lire(A)
    lire(B)
    affiche(A, B) # invocation
    lire(C)
    lire(D)
    affiche(C, D) # invocation
    ...
Fin
```



exemple avec modification locale de paramètre

```

Algorithme testPrc12 :
Variable n1, n2 : Entier
...
Procédure décompte(x : Entier) :           # prototype, signature
Début
  tantQue non (x = 0)
    écrire(x)
    x ← x - 1                               # modification de la variable locale x
  finTQ
Fin;
...
Début
  ...
  n1 ← 15
  n2 ← 7
  décompte(n1) # passage par copie/valeur; n1 reste inchangé après invocation
  décompte(n2) # idem
  ...
Fin

```

## b) ... AVEC MODIFICATION DE DONNÉES

une déclaration de procédure avec passage de paramètres par référence/adresse utilise le modèle général suivant :

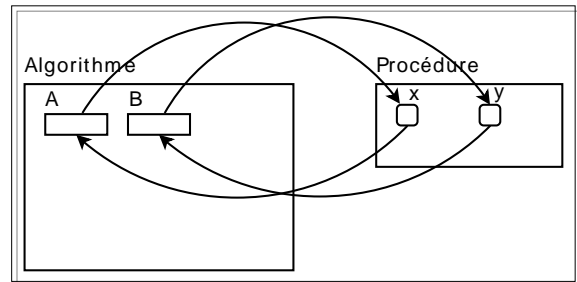
```

Procédure nomDeLaProcédure (ref nom et type des paramètres) :
Début
...
# corps de la procédure
Fin

```

chaque paramètre correspondant à une variable de l'appelant qui doit être modifiée par la procédure est précédée par le mot ref (référence à ...)

l'exemple typique d'une procédure avec paramètre d'entrée est la procédure de lecture lire(...) jouant le rôle d'affectation externe : on lui transmet l'adresse de la variable réceptrice; de sorte que lire(v) a pour signification "mettre ce qui a été introduit au clavier en mémoire à l'adresse correspondant à la variable v"



on évitera habituellement d'utiliser une telle procédure pour lui faire jouer le rôle naturellement réservé à une fonction

```

Algorithme testPrc21 :
Variable A : entier
...
Procédure cube(ref x : Entier)
Début
  x ← x * x * x
Fin
...
Début
  ...
  lire(A)
  cube(A)           # modifie A
  ...
Fin

```

```

Algorithme testPrc21 :
Variable A : entier
...
Fonction cube(x : Entier)
Début
  renvoie(x * x * x)
Fin
...
Début
  ...
  lire(A)
  A ← cube(A)
  ...
Fin

```

par contre, comme les fonctions ne peuvent retourner qu'une seule valeur, on utilisera les procédures quand plusieurs résultats sont attendus ou quand il faut modifier plusieurs variables

l'exemple le plus représentatif (et que l'on retrouvera dans les tris de tableaux) est l'échange de valeurs entre deux variables déjà évoqué

```

Algorithme testPrc22 :
Variable A, B, C, D, tmp : Entier
...
Début
...
lire(A); lire(B)
lire(C); lire(D)
tmp ← A
A ← B
B ← tmp
tmp ← C
C ← D
D ← tmp
...
Fin

```

```

Algorithme testPrc22 :
Variable A, B, C, D : Entier
...
Procédure échanger(ref x, ref y : Entier)
Variable tmp : Entier
Début
    tmp ← x
    x ← y
    y ← x
Fin;
...
Début
...
lire(A); lire(B)
lire(C); lire(D)
échanger(A, B)
échanger(C, D)
...
Fin

```

Fonctions, procédures avec paramètres par copie/valeur et procédures avec paramètres par référence/adresse prendront tout leur sens sur les structures de données ... nous y revenons dans et après les chapitres qui leurs sont consacrés ...

## 4.5. SOUPLESSE ET ABSTRACTION : LA SURCHARGE

En quelques mots, juste pour introduire le sujet ...<sup>49</sup>

Les concepts de programmation impérative procédurale (nous regrouperons désormais sous ce terme unique les notions de procédure et de fonction) présentés dans ce cours sont largement inspirés par les langages "statiques fortement typés" (comme le Pascal, par exemple) ...

L'avantage principal de la démarche est l'apprentissage de la rigueur : tout (variables globales et locales, paramètres des procédures et fonctions, valeur de retour de fonction) doit être systématiquement déclaré (nommé) et typé; l'inconvénient est un certain manque de souplesse ... reprenons la fonction validatrice écrite précédemment :

```

Algorithme saisieNombres
Constante vMin1 = 10, vMax1 = 100, vMin2 = 1, vMax2 = 20
Variable nombre1, nombre2 : Entier
...
Fonction lireEntier(min, max : Entier) : Entier
Variable n : Entier
Début
    écrire("introduire un nombre entre ", min, " et ", max, " : ")
    lire(n)
    tantQue non (n entre min et max) faire
        écrire("valeur hors intervalle, recommencez : ")
        lire(n)
    finTQ
    renvoie(n)
Fin
...
Début
    ...
    nombre1 ← lireEntier(vMin1, vMax1)
    nombre2 ← lireEntier(vMin2, vMax2)
    ...
Fin

```

et ajoutons au problème la nécessité de saisir un nombre réel compris entre -50.0 et +50.0 (p.ex. des températures) : il faudrait ajouter une seconde fonction pour valider des réels !

```

Constante vMin3 = -50.0, vMax3 = 50.0
Variable nombre3 : Réel
...
Fonction lireRéel(min, max : Réel) : Réel
Variable n : Réel
Début
    écrire("introduire un nombre entre ", min, " et ", max, " : ")
    lire(n)
    tantQue non (n ≥ min et n ≤ max) faire
        écrire("valeur hors intervalle, recommencez : ")
        lire(n)
    finTQ
    renvoie(n)
Fin

Début
    ...
    nombre3 ← lireRéel(vMin3, vMax3)
    ...
Fin

```

... on imagine bien que cela ne va pas dans le sens de la simplicité ni de la souplesse ...

<sup>49</sup> qui sera développé dans le cours d'Organisation et Structure de Données, et expérimenté via la notion de Programmation générique dans le cours de Langage C

Dès lors, pour que le typage fort ne soit pas un obstacle, on admet généralement (c'est le cas en Pascal, en C++<sup>50</sup>, comme dans la plupart des langages actuels) le principe de la surcharge (overloading) des fonctions paramétrées (et des procédures paramétrées, le principe est identique) :

rappelons d'abord qu'on entend par signature d'une fonction (d'une procédure) : c'est la ligne d'en-tête comportant

- le nom de la fonction (de la procédure)
- le nombre, l'ordre et le type des paramètres (leur nom n'a pas d'importance)
- le type de la valeur de retour (uniquement pour les fonctions, bien sûr)

*des fonctions (des procédures) différentes peuvent porter le même nom pour autant que leur signature soit différente*

```

Algorithme saisieNombres
Constante eMin = 10, eMax = 100, oui = 'o', non = 'n',
          rMin = -50.0, rMax = 50.0
Variable eNombre : Entier, réponse : Caractère, rNombre : Réel
...
Fonction fLire(min, max : Entier) : Entier
Variable n : Entier
Début
    écrire("introduire un nombre entre ", min, " et ", max, " : ")
    lire(n)
    tantQue non (n entre min et max) faire
        écrire("valeur hors intervalle, recommencez : ")
        lire(n)
    finTQ
    renvoie(n)
Fin

Fonction fLire(min, max : Réel) : Réel
Variable n : Réel
Début
    écrire("introduire un nombre entre ", min, " et ", max, " : ")
    lire(n)
    tantQue non (n entre min et max) faire
        écrire("valeur hors intervalle, recommencez : ")
        lire(n)
    finTQ
    renvoie(n)
Fin

Fonction fLire(c1, c2 : Caractère) : Caractère
Variable c : Caractère
Début
    écrire("voulez-vous continuer (", c1, "/", c2, ") ? ")
    lire(c); c ← minuscule(c)
    tantQue non (c = c1 ou c = c2) faire
        écrire("réponse incorrecte, recommencez : ")
        lire(c); c ← minuscule(c)
    finTQ
    renvoie(c)
Fin

...
Début
    ...
    eNombre ← fLire(eMin, eMax)
    réponse ← fLire(oui, non)
    rNombre ← fLire(rMin, rMax)
    ...
Fin

```

<sup>50</sup> pas en C, malheureusement !

## 4.6. CONCRÈTEMENT ...

Dans un chapitre précédent (3.1 *Les Instructions d'Action*, page 36), comportant une introduction à la notion de procédure, un même exemple (petit établissement de facture) était rédigé dans quelques langages impératifs procéduraux 'classiques' (Fortran , Cobol, Pascal, C) ... on avait pu y découvrir l'usage des variables globales

Nous reprenons à présent ces mêmes exemples, mais pour y illustrer l'usage des fonctions et des procédures paramétrées ... au lecteur de comparer avec les exemples précédents ...

### Cobol

Dans la mesure où ce langage (même dans ses versions récentes) n'offre aucune possibilité de création de (vraie) fonction et que l'usage de procédures paramétrées est assez lourd et complexe, nous en resterons là avec ce langage : les procédures sont natives dans le langage sous forme de 'paragraphes' et l'échange de données s'effectue via les variables globales

```
IDENTIFICATION DIVISION.
PROGRAM-ID.                                FACTURE.
*
*** déclaration des données *****
*
DATA DIVISION.
WORKING-STORAGE SECTION.
*
* pseudo-constante
*
77 TVA                                PIC 9V99 VALUE 0.21.
*
* variables globales
*
77 QUANTITE                            PIC 999.
77 PRIXUNIT                            PIC 999V99.
77 PRIXHTVA                            PIC 9999V99.
77 PRIXTVAC                            PIC 9999V99.
*
*** sequence principale *****
*
PROCEDURE DIVISION.
MAIN SECTION.
    PERFORM SAISIR.
    PERFORM CALCULER.
    PERFORM AFFICHER.
    EXIT PROGRAM.
    STOP RUN.
*
*** procedures
*
SAISIR.
    DISPLAY "prix unitaire du produit (en euro) ? ".
    ACCEPT PRIXUNIT.
    DISPLAY "quantité vendue ? ".
    ACCEPT QUANTITE.

CALCULER.
    COMPUTE PRIXHTVA = PRIXUNIT * QUANTITE.
    COMPUTE PRIXTVAC = PRIXHTVA * (1.0 + TVA).

AFFICHER.
    DISPLAY "prix HTVA : " PRIXHTVA " euro".
    DISPLAY "prix TVAC : " PRIXTVAC " euro".
*
*****
```

## Fortran

Dès les premières versions majeures du langage (Fortran IV, 1962), les concepts de procédure (SUBROUTINE), de fonctions (FUNCTION), de paramètres et de variables globales et locales sont implémentés.

Dans la mesure où les variables globales ne sont accessibles par les procédures et les fonctions qu'à travers une déclaration d'un bloc de données partagé (COMMON), le passage de paramètres offre une souplesse et une formalisation bienvenues : les paramètres sont transmis par copie/valeur à une fonction et par référence/adresse à une procédure : leurs types ne sont pas spécifiés dans la signature, mais redéclarés explicitement

```

PROGRAM FACTURE
c *** déclaration des variables
  INTEGER QUANT
  REAL TVA, PRUNIT, PRHTVA, PRTVAC
c *** programme principal
  TVA = 0.21
  CALL SAISIR(QUANT, PRUNIT)
  CALL CALCULER(QUANT, PRUNIT, TVA, PRHTVA, PRTVAC)
  CALL AFFICHER(PRHTVA, PRTVAC)
END

c *** procédures et fonctions
  SUBROUTINE SAISIR(Q, PU)
c *** procédure de saisie de la quantité Q et du prix unitaire PU
  INTEGER Q
  REAL PU
11  FORMAT('prix unitaire du produit (en euro) ? ')
12  FORMAT('quantite vendue ? ')
21  FORMAT(I2)
22  FORMAT(F5.2)
  WRITE(6,11)
  READ(5,21) Q
  WRITE(6,12)
  READ(5,22) PU
END

  REAL FUNCTION CALCPRHTVA(Q, PU)
c *** fonction qui calcule et renvoie le prix hors tva ... partir
c *** de la quantité Q et du prix unitaire PU passés comme paramètre
  INTEGER Q
  REAL PU
  CALCPRHTVA = Q * PU
  RETURN
END

  REAL FUNCTION CALCPRTVAC(PRH, T)
c *** fonction qui calcule et renvoie le prix tva comprise ... partir
c *** du prix hors tva PRH et de la tva T passés comme paramètres
  REAL PRH, T
  CALCPRTVAC = PRH * (1 + T)
  RETURN
END

  SUBROUTINE CALCULER(Q, PU, T, PRH, PRC)
c *** procédure de calcul du prix hors tva PRH et tva comprise PRC
c *** ... partir de la quantité Q, du prix unitaire PU et de la tva T
  INTEGER Q
  REAL PU, T, PRH, PRC
  PRH = CALCPRHTVA(Q, PU)
  PRC = CALCPRTVAC(PRH, TVA)
END

  SUBROUTINE AFFICHER(PRH, PRC)
c *** procédure affichage du prix hors tva PRH et tva comprise PRC
  REAL PRH, PRC
13  FORMAT('prix HTVA : ', F6.2, ' euro')
14  FORMAT('prix TVAC : ', F6.2, ' euro')
  WRITE(6,13) PRH
  WRITE(6,14) PRC
END

```

## Pascal

C'est – faut-il s'en étonner – le langage le plus proche de la philosophie du pseudo-code. Les fonctions sont de 'vraies' fonctions (avec paramètres passés par copie/valeur et ont un statut de variables locales dans le corps de la fonction); les procédures peuvent recevoir les paramètres par copie/valeur (protection garantie) ou par référence/adresse (pour pouvoir modifier les variables blogales)

```

program facture;
uses crt;
const tva = 0.21;

{ *** variables globales ***** }

var quantite : integer;
    prix_unitaire, total_ht, total_ttc : real;

{ *** procedures ***** }

procedure saisir(var q : integer; var pu : real);
begin
    write('prix unitaire de l'article (en euro) ? ');
    readln(pu);
    write('combien d'articles voulez-vous      ? ');
    readln(q);
end;

function calc_htva(q : integer; pu : real) : real;
begin
    calc_htva := q * pu;
end;

function calc_tvac(htva : real) : real;
begin
    calc_tvac := htva * (1 + tva);
end;

procedure calculer(q : integer; pu : real; var htva, tvac : real);
begin
    htva := calc_htva(q, pu);
    tvac := calc_tvac(htva);
end;

procedure afficher(htva, tvac : real);
begin
    writeln;
    writeln('total ht   : ',htva:6:2, ' euro');
    writeln('total ttc : ',tvac:6:2, ' euro');
end;

procedure terminer;
begin
    writeln;
    write('appuyez sur une touche pour continuer ... ');
    readkey;
end;

{ *** sequence principale ***** }

begin
    saisir(quantite, prix_unitaire);
    calculer(quantite, prix_unitaire, total_ht, total_ttc);
    afficher(total_ht, total_ttc);
    terminer;
end.

```



C

Il y a une influence certaine du Fortran (et d'autres langages précurseurs) en C. (rédaction parfois en 'sens inverse' : type puis noms des variables, etc ...)

La caractéristique fondamentale du C est de ne proposer qu'un seul outil hybride : la "fonction C" avec laquelle le programmeur rédigera soit de 'vraies' fonctions, soit de 'vraies' procédures, soit des fonctions/procédures (?) ou des procédures/fonctions (?) : une chose est sûre, comme toujours en C (et contrairement au Pascal), c'est au programmeur de faire ses choix (et d'en assumer les conséquences); les paramètres sont toujours passés par copie/valeur et le passage 'par adresse' demande du programmeur d'utiliser des opérateurs spécifiques (et la manipulation des adresses des données : les 'pointeurs', cauchemars des débutants)

```
#include <stdio.h>           // librairie
#include <stdlib.h>          // librairie

#define TAUX TVA 0.21       // constante

// variables globales
int quantiteVendue;
double prixUnitaire, prixHTva, prixTvaC;

// prototypes des fonctions & procedures
void saisir(int*, double*);
double calcHtva(int, double);
double calcTvac(int, double);
void calculer(int, double, double*, double*);
void afficher(double, double);

// séquence principale
int main() {
    saisir(&quantiteVendue, &prixUnitaire);
    calculer(quantiteVendue, prixUnitaire, &prixHTva, &prixTvaC);
    afficher(prixHTva, prixTvaC);

    system("pause");
    return 0;
}

// procédures & fonction
void saisir(int* q, double* pu) { // procédure
    system("cls");
    printf("prix unitaire du produit (en €) ? ");
    scanf("%lf", pu);
    printf("quantité vendue ? ");
    scanf("%d", q);
}

double calcHtva(int q, double pu) { // fonction
    return q * pu;
}
double calcTvac(int q, double pu) { // fonction
    return calcHtva(q, pu) * (1.0 + TAUX TVA);
}

void calculer(int q, double pu, double* htva, double* tvac) { // procédure
    *htva = calcHtva(q, pu);
    *tvac = calcTvac(q, pu);
}

void afficher(double htva, double tvac) { // procédure
    printf("prix HTVA : %f €\n", htva);
    printf("prix TVAC : %f €\n", tvac);
}
```

## 4.7. EXERCICES

- 1) Écrire une fonction qui prenne en arguments (paramètres) d'entrée deux nombres réels H et R et qui renvoie la surface du cylindre de hauteur H et de rayon R
- 2) Ecrire une fonction qui calcule  $x^n$  avec comme arguments x et n deux entiers et telle que n est positif ou nul ; intégrez-la dans un algorithme principal qui fasse les validations nécessaires avant l'appel de la fonction
- 3) Le cours de mathématiques introduit le nombre e comme  $\lim_{x \rightarrow \infty} (1 + \frac{1}{x})^x$ 
  - Écrivez une fonction dont la signature est la suivante : *Fonction e(x : Entier) : Réel*
  - Placez sa définition (et son contenu) au bon endroit au sein d'un algorithme principal
  - Invoquez-la en lui passant des valeurs de x telles que 10, 100, 1000 ... (*exercice à suivre en Pascal ultérieurement ...*)

- 4) On suppose déclarées les procédures, fonctions et variables suivantes :

```

variable x, y, z, t : Entier
Procédure p(x : Logique, ref s : Entier)
Fonction f(s : Logique) : Entier

```

dans ce contexte, quelles sont les instructions invalides parmi celles ci-dessous ?

Précisez pour quelles raisons

1.  $p(\text{vrai}, z)$
2.  $z \leftarrow f(z)$
3. *Si*  $p(x, z) = z$  *Alors*  $p(x, z)$  *finSi*
4.  $f(\text{vrai})$
5.  $x \leftarrow f(\text{vrai})$
6.  $p(\text{vrai}, 3)$
7.  $p(2, z)$
8.  $p(x, f(z))$
9.  $p(f(x), z)$
10.  $z \leftarrow f(x = 3)$
11.  $x \leftarrow f(x) = 3$
12.  $t \leftarrow f(z = 3)$

- 5) On suppose déclarées la procédure et les variables suivantes :

```

Variable x, y, z, t : Entier
Procédure q(x : Entier, ref y : Entier, ref z : Entier)
  Début
  ...
  Fin

```

En supposant les variables initialisées par  $\{x \leftarrow 1; y \leftarrow 2; z \leftarrow 3\}$  indiquez la valeur de ces variables après chacune des instructions suivantes ? (Dans chacune des sous-questions, on supposera que préalablement on a  $\{x \leftarrow 1; y \leftarrow 2; z \leftarrow 3\}$ .)

1.  $q(1, y, x)$
2.  $q(2 * x, y, z)$
3.  $q(x + y + z, z, x)$

- 6) Écrire une fonction dont la signature est la suivante :

*Fonction formatDate(j, m, a : Entier) : Texte*

- elle accepte en entrée trois entiers correspondant respectivement au jour, mois et année d'une date (ces données sont supposées correctes)
- elle fournit comme valeur de sortie cette date sous forme d'un texte normalisé sur 10 caractères jj/mm/aaaa
  - p.ex. *formatDate(26, 10, 2010)* renvoie *26/10/2010*
  - p.ex. *formatDate(3, 2, 2011)* renvoie *03/02/2011*

- 7) Modifier ensuite cette fonction pour lui donner la signature suivante :

*Fonction formatDate(j, m, a : Entier, mode : Caractère) : Texte*

le paramètre *mode* (1 caractère supposé correct valant soit 'n' soit 'm') précise cette fois la forme du format de sortie :

- si *mode* = 'n', la fonction *formatDate(3, 2, 2011, 'n')* renvoie la date sous forme d'un texte normalisé sur 10 caractères jj/mm/aaaa *03/02/2010*, comme précédemment
- si *mode* = 'm', la fonction *formatDate(3, 2, 2011, 'm')* renvoie la date sous forme d'un texte normalisé sur 11 caractères jj mmm aaaa *03 fev 2010* (trois premières lettres du libellé du mois)

si nécessaire, il est recommandé de décomposer ce problème en procédures et/ou fonctions utiles (une fonction peut évidemment invoquer une autre fonction et/ou une autre procédure ...)

- 8) En vous souvenant des algorithmes que vous avez (auriez ? ☺) écrits précédemment sur la détermination des dates (et sachant que le 1<sup>er</sup> janvier 1900 était un lundi), ajoutez à la fonction *formatDate* la fonctionnalité suivante :

- si *mode* = 't', la fonction *formatDate(3, 2, 2011, 't')* renvoie la date sous forme d'un texte normalisé sur 15 caractères jjj jj mmm aaaa *jeu 03 fev 2010* (trois premières lettres du libellé du jour, trois premières lettres du libellé du mois)

cette fois l'écriture de fonctions et/ou procédures supplémentaires est indispensable

- 9) La fête de Pâques est une fête mobile dont la date est assez difficile à déterminer (prévoir); on appelle "comput ecclésiastique" la(les) formule(s) qui permet(tent) de déterminer la date de Pâques pour une année donnée<sup>51</sup>

La date de Pâques correspond au 1<sup>er</sup> dimanche après la première pleine lune qui suit l'équinoxe de printemps. Soient les données suivantes :

- a = année modulo 19
- b = année modulo 4
- c = année modulo 7
- d = (19 \* a + 24) modulo 30
- e = (2 \* b + 4 \* c + 6 \* d + 5) modulo 7

Avec ces cinq expressions, on détermine  $n = 22 + d + e$ , où **n** est le nombre de jours à ajouter au 1<sup>er</sup> mars pour obtenir la date du dimanche de Pâques.

Rédigez une fonction qui calcule la date de Pâques à partir de l'année; sa signature est :

*Fonction dimPâques(an : Entier) : Texte*

elle fournit comme valeur de sortie cette date sous forme d'un texte normalisé sur 10 caractères jj/mm/aaaa

<sup>51</sup> [http://fr.wikipedia.org/wiki/Calcul\\_de\\_la\\_date\\_de\\_Pâques](http://fr.wikipedia.org/wiki/Calcul_de_la_date_de_Pâques)



## 5. LES DONNÉES (II)

### 5.1. LIMITE DES DONNÉES SCALAIRES

Limiter les possibilités d'utilisation de données par les programmes informatiques aux seules données dites 'scalaires' (Entier, Réel, Caractère, Logique ...) s'est rapidement révélé insuffisant.

En effet, généralement, on souhaite informatiser (=automatiser) des problématiques comportant de vastes ensembles ou 'collections' de données, et ne disposer pour ce faire que de variables scalaires (1 donnée = 1 variable) rend la rédaction de la solution non seulement lourde et pénible, mais à la limite du possible; qu'on en juge par ce tout petit exemple :

on souhaite informatiser la gestion des résultats d'une interrogation à laquelle sont soumis des étudiants (ici seulement 5 pour illustrer le propos et ménager le rédacteur de ces lignes, au lecteur d'imaginer ce qu'il faudrait faire pour un cas réaliste de disons 100 étudiants); on veut pouvoir encoder les cotes, les afficher, mais surtout les trier avant affichage : il faut donc les conserver en mémoire (on les encode toutes, on les trie et on affiche le résultat de ce classement)

on pourrait commencer par déclarer les données comme suit

```
Algorithme gestionCotes :
constante cMin = 0.0, cMax = 20.0                # pour la validation
Variable cote1, cote2, cote3, cote4, cote5 : Réel  # pour 5 étudiants
```

puis, on écrirait une procédure d'encodage/validation ...

```
Procédure encodage :
Début
  Répéter
    écrire("entrez la cote 1 (", cMin, "..", cMax, ") : ")
    lire(cote1)
  Jusqu'à cote1 entre cMin et cMax
  Répéter
    écrire("entrez la cote 2 (", cMin, "..", cMax, ") : ")
    lire(cote2)
  Jusqu'à cote2 entre cMin et cMax
  Répéter
    écrire("entrez la cote 3 (", cMin, "..", cMax, ") : ")
    lire(cote3)
  Jusqu'à cote3 entre cMin et cMax
  Répéter
    écrire("entrez la cote 4 (", cMin, "..", cMax, ") : ")
    lire(cote4)
  Jusqu'à cote4 entre cMin et cMax
  Répéter
    écrire("entrez la cote 5 (", cMin, "..", cMax, ") : ")
    lire(cote5)
  Jusqu'à cote5 entre cMin et cMax
Fin
```

on pourrait objecter qu'on a ici volontairement choisi la manière la plus stupide d'écrire cela, et qu'une écriture plus procédurale et plus légère pourrait s'envisager :

```
Algorithme gestionCotes :
constante cMin = 0.0, cMax = 20.0                # pour la validation
Variable cote1, cote2, cote3, cote4, cote5 : Réel  # pour 5 étudiants
        num : Entier, cote : Réel                # pour saisie

Procédure lireCote :
Début
  Répéter
    écrire("entrez la cote ", num, " (", cMin, "..", cMax, ") : ")
    lire(cote)
  Jusqu'à cote entre cMin et cMax
Fin
```

```

Procédure encoderNombres :
Début
    num ← 1; lireCote; cote1 ← cote
    num ← 2; lireCote; cote2 ← cote
    num ← 3; lireCote; cote3 ← cote
    num ← 4; lireCote; cote4 ← cote
    num ← 5; lireCote; cote5 ← cote
Fin

```

de fait ... c'est un peu plus 'automatisé', mais il reste 1 ligne (de 3 instructions) à écrire pour chacune des cotes ... et tant qu'à faire, si l'on cherche la meilleure expression du problème, profitons du chapitre précédent (procédures et fonctions paramétrées) pour écrire plutôt :

```

Algorithme gestionCotes :
constante cMin = 0.0, cMax = 20.0                # pour la validation
Variable cote1, cote2, cote3, cote4, cote5 : Réel # pour 5 étudiants

Fonction lireCote(n : Entier) : Réel :           # saisie et renvoi d'une cote valide
# paramètre d'entrée n : le numéro à spécifier dans l'invite
# valeur de sortie : une cote entre cMin et cMax
Variable cote : Réel                             # variable locale pour la saisie
Début
    Répéter
        écrire("entrez la cote ", n, " (", cMin, "..", cMax, ") :")
        lire(cote)
    Jusqu'à cote entre cMin et cMax               # ici, cote ok
    renvoie(cote)                                # retour au demandeur
Fin

Procédure encoderCotes :
Début
    cote1 ← lireCote(1)
    cote2 ← lireCote(2)
    cote3 ← lireCote(3)
    cote4 ← lireCote(4)
    cote5 ← lireCote(5)
Fin

```

... mais toujours une ligne par cote évidemment (mais toute simple : avez-vous remarqué qu'une fonction permet de généraliser l'opérateur d'affectation ?); l'utilisation d'une boucle commence à nous démanger, mais il reste cette partie gauche de l'affectation : des variables scalaires ... portant toutes des noms différents !

passons à la suite : les cinq cotes sont introduites en mémoire, il faut à présent les classer (disons par ordre croissant) avant de pouvoir les afficher ... cette dernière opération pourra s'écrire (ici parce qu'il n'y a que cinq cotes) assez facilement avec une procédure que l'on pourra réutiliser dans l'algorithme principal, une fois avant le classement et une fois après ...

```

Procédure montrerCotes :
Début
    écrire(cote1, " ", cote2, " ", cote3, " ", cote4, " ", cote5)
Fin

Début                                     # séquence principale
    encoderCotes
    montrerCotes
    trierCotes                             # encore à écrire
    montrerCotes
Fin

```

Classer des variables : essayez donc cela à grande échelle (et même ici avec seulement cinq variables)... avec pour seule arme des Si ... Alors ... Sinon ... ! nous allons tricher un tout petit peu et appliquer un principe que l'on retrouvera plus loin lors de l'étude des tris : on va rechercher la plus petite valeur des cinq variables et placer cette valeur (par échange) en première place, puis on fera de même avec les variables 2 à 5, puis avec les variables 3 à 5 ...

```

Procédure trierCotes :
Variable tmp : Réel          # variable locale pour l'échange
Début
  Si cote2 < cote1 Alors
    tmp ← cote1; cote1 ← cote2; cote2 ← tmp finSi
  Si cote3 < cote1 Alors
    tmp ← cote1; cote1 ← cote3; cote3 ← tmp finSi
  Si cote4 < cote1 Alors
    tmp ← cote1; cote1 ← cote4; cote4 ← tmp finSi
  Si cote5 < cote1 Alors
    tmp ← cote1; cote1 ← cote5; cote5 ← tmp finSi
  # ici cote1 contient la plus petite cote des 5
  Si cote3 < cote2 Alors
    tmp ← cote2; cote2 ← cote3; cote3 ← tmp finSi
  Si cote4 < cote2 Alors
    tmp ← cote2; cote2 ← cote4; cote4 ← tmp finSi
  Si cote5 < cote2 Alors
    tmp ← cote2; cote2 ← cote5; cote5 ← tmp finSi
  # ici cote2 contient la plus petite cote des 4 restantes
  Si cote4 < cote3 Alors
    tmp ← cote3; cote3 ← cote4; cote4 ← tmp finSi
  Si cote5 < cote3 Alors
    tmp ← cote3; cote3 ← cote5; cote5 ← tmp finSi
  # ici cote3 contient la plus petite cote des 3 restantes
  Si cote5 < cote4 Alors
    tmp ← cote4; cote4 ← cote5; cote5 ← tmp finSi
  # cote4 contient la plus petite cote des 2 restantes
Fin

```

☹ on était prévenus que ce serait galère ... le seul endroit où l'on peut économiser dans l'écriture, c'est l'échange ... profitons-en pour introduire une procédure paramétrée avec passage des variables par référence : c'est un grand classique que l'on retrouvera plus loin

```

Procédure échangerCotes (ref c1 : Réel, ref c2 : Réel) :
# paramètres d'entrée c1, c2 : les deux variables dont il faut permuter le contenu
# comme elles sont modifiées, elles sont passées par référence
Variable tmp : Réel          # variable locale pour l'échange
Début
  tmp ← c1; c1 ← c2; c2 ← tmp
Fin

```

ainsi, "il n'y a plus qu'à" récrire ...

```

Procédure trierCotes :
Début
  Si cote2 < cote1 Alors échangerCotes(cote1, cote2) finSi
  Si cote3 < cote1 Alors échangerCotes(cote1, cote3) finSi
  Si cote4 < cote1 Alors échangerCotes(cote1, cote4) finSi
  Si cote5 < cote1 Alors échangerCotes(cote1, cote5) finSi
  # cote1 contient la plus petite cote des 5
  Si cote3 < cote2 Alors échangerCotes(cote2, cote3) finSi
  Si cote4 < cote2 Alors échangerCotes(cote2, cote4) finSi
  Si cote5 < cote2 Alors échangerCotes(cote2, cote5) finSi
  # cote2 contient la plus petite cote des 4 restantes
  Si cote4 < cote3 Alors échangerCotes(cote3, cote4) finSi
  Si cote5 < cote3 Alors échangerCotes(cote3, cote5) finSi
  # cote3 contient la plus petite cote des 3 restantes
  Si cote5 < cote4 Alors échangerCotes(cote4, cote5) finSi
  # cote4 contient la plus petite cote des 2 restantes
Fin

```

☹ très, très limite tout de même ...

Examinons un autre problème :

on souhaite informatiser la gestion des renseignements signalétiques de diverses personnes (étudiants et enseignants par exemple; ici on se contentera d'une de 'chaque sorte'); parmi ces renseignements figurent les noms et prénoms, les dates de naissance, les adresses (rue, codePostal, Ville), etc ...; les étudiants et les enseignants ont en plus une valeur réelle en commun, mais pour les premiers il s'agit du résultat (pourcentage) sur l'ensemble de l'année, alors que pour les seconds, il s'agit du salaire ...; pour les besoins de l'application, toutes ces informations doivent être encodées et maintenues en mémoire pour un usage ultérieur ...

A nouveau c'est le fait d'être confronté à un (potentiellement) large ensemble de données qui pose problème, tant à la déclaration qu'à l'utilisation de ces données :

on pourrait commencer par déclarer les données comme suit

```
Algorithme gestionPersonnes :
Variable nomEtu, nomProf, prenomEtu, prenomProf, rueEtu, rueProf : Texte
      codPosEtu, codPosProf : Entier      # pas super ...
      villeEtu, villeProf : Texte
      jourEtu, jourProf, moisEtu, moisProf, annéeEtu, annéeProf : Entier
      salaire : Réel; résultat : Réel
      estEtu, estProf : Logique
```

mais pour encoder les renseignements relatifs à un étudiant (ou à un enseignant), il faut écrire une procédure spécifique : (ni les invites, ni les validations ne sont représentées ci-dessous)

```
Procédure encoderEtu :
Début
  lire(nomEtu); lire(prenomEtu)
  lire(rueEtu); lire(codPosEtu); lire(villeEtu)
  lire(jourEtu); lire(moisEtu); lire(annéeEtu)
  lire(résultat)
  estEtu ← vrai
Fin

Procédure encoderProf :
Début
  lire(nomProf); lire(prenomProf)
  lire(rueProf); lire(codPosProf); lire(villeProf)
  lire(jourProf); lire(moisProf); lire(annéeProf)
  lire(salaire)
  estProf ← vrai
Fin
```

première remarque : on a affaire à un ensemble 'varié' de variables, et on ne doit qu'à la bonne volonté du programmeur (parce qu'il a suffixé ses noms de variables avec Etu ou avec Prof) d'y voir un semblant d'ordre (par contre les variables salaire et résultat n'ont pas d'appartenance a priori)

deuxième remarque : chaque (sous-)ensemble de données est disparate : on y trouve aussi bien des données de type Texte, Entier et Réel et il contient lui-même un (sous-)ensemble : les variables jour..., mois..., année... font évidemment penser à une date (de naissance ou d'autre chose, inscription, prise de fonction ... ce n'est pas précisé), et il faudrait encore préfixer ou suffixer ces noms pour qu'ils deviennent plus explicites quant à leur rôle ...)



Les deux exemples ont présenté les principales difficultés rencontrées avec de larges ensembles de données :

- dans le premier cas, il s'agit de données de même nature (des cotes sous forme de nombres réels) sur lequel on souhaiterait utiliser des boucles, mais on se heurte au nommage des variables ... *"ahhh ... si on pouvait écrire une boucle de nommage ... en concaténant un même nom (p.ex. cote) avec la valeur d'un compteur incrémenté à chaque tour de boucle ..."*
- dans le second cas, il y a beaucoup de variables, possédant des noms et des types différents, qui constituent en fait des aspects complémentaires d'une même 'entité' (étudiant, enseignant) *"ahhh ... si on pouvait exprimer – sans devoir utiliser un subterfuge comme le suffixage des noms - : lire(le nom d'un prof), écrire(la ville d'un étudiant) ..."*
- et puis, bien entendu, il faut imaginer pouvoir combiner les deux problématiques : un ensemble de plusieurs étudiants, un étudiant disposant d'un ensemble de plusieurs cotes, un ensemble d'étudiants disposant chacun d'un ensemble de cotes, etc, etc ...

Ceci sonne le glas de l'utilisation des seules variables scalaires en algorithmique et en programmation; il faut disposer (côté données) de quelque chose de 'plus haut niveau' pour pouvoir envisager aller plus loin ... et on se doute bien que (côté algorithmique) il faudra des outils adaptés à ce 'quelque chose')

Ce 'quelque chose', ce sont les "structures de données" qui vont permettre de rassembler sous un même nom unique un ensemble de valeurs (contrairement aux données scalaires qui n'en contiennent jamais qu'une seule)

Dès l'apparition des premiers langages informatiques, deux familles de structures de données ont fait leur apparition : les Tableaux (Fortran, langage 'scientifique', 1953) et les Enregistrements (Cobol, langage 'de gestion', 1959).

Ces structures vont servir de 'constructeurs' de base pour l'organisation et la structure de données plus complexes<sup>52</sup> et faire littéralement 'exploser' les possibilités offertes aux analystes et programmeurs.

## 5.2. LA STRUCTURE D'ENREGISTREMENT

### a) PREMIER CONTACT

L'Enregistrement est une structure de données qui offre l'hétérogénéité, ou inhomogénéité (c-à-d qu'il permet de rassembler sous un même nom un ensemble de données, chacune conservant un nom propre et, possédant (éventuellement) un type différent).

Provenant des besoins exprimés durant les années 1960 par le monde de la gestion, très demandeur d'informatisation, les enregistrements annoncent et accompagnent d'ailleurs eux-mêmes une autre structure de données utilisée pour assurer la "persistance des données", et en particulier pour le stockage permanent des données sur supports externes : les fichiers<sup>53</sup>.

On peut percevoir l'Enregistrement comme la réunion au sein d'une seule et même structure d'un ensemble de variables participant chacune à une partie de la description d'une même entité; la structure va permettre – une fois encore, c'est toujours le même objectif qui est poursuivi – de 'coller' au plus près au problème posé, d'augmenter le niveau d'abstraction et de fournir au programmeur les outils techniques adaptés.<sup>54</sup>

<sup>52</sup> cfr. le cours spécifique consacré aux "Organisations et Structures de Données" au second semestre ...

<sup>53</sup> idem

<sup>54</sup> l'évolution 'naturelle' de la structure d'enregistrement a conduit à la composante 'données' de l'approche 'objet' (classes, instances et propriétés), tandis que dans le même temps, l'évolution de la partie algorithmique (procédures et fonctions) a conduit à la composante 'outils' de cette même approche (méthodes)

Exemple : reprenons notre problématique de gestion de personnes; on peut (re)partir de ce que l'on connaît : les variables scalaires

```
Variable nom, prénom, adresse, ville : Texte
        codePostal : Entier
        annéeNaissance, moisNaissance, jourNaissance : Entier
```

... mais rien n'indique a priori (sauf dans la tête de l'informaticien, qui sait, lui, en principe ! et dans les commentaires qu'il aurait la bonté de rédiger) que cet ensemble de variables forme un tout cohérent, relatif en fait à un seul et même sujet : une personne

La structure d'Enregistrement ("Record" en anglais) va permettre de rassembler les multiples et différentes 'facettes' de la description d'une personne dans une structure unique, via la déclaration (provisoire) suivante :

```
Variable personne : Enregistrement
        nom           : Texte
        prénom        : Texte
        annéeNaissance : Entier
        moisNaissance : Entier
        jourNaissance  : Entier
        adresse       : Texte
        codePostal     : Entier
        ville         : Texte
finEnr
```

On voit apparaître :

- un nom symbolique (un identificateur donné par le programmeur) pour l'ensemble de la structure (ici *personne*); ce nom sera également qualifié de nom générique (on y reviendra)
- et – comme pour les structures de contrôle - deux marqueurs de début et de fin délimitant la déclaration de la composition de l'enregistrement (**Enregistrement**, **finEnr**<sup>55</sup>);
- entre ces deux marqueurs, des déclarations nommées de données 'scalaires'<sup>56</sup> pour chaque 'facette' de l'entité décrite (c.-à-d. : un nom et un type scalaire associé); la terminologie officielle pour 'composant' ou 'facette', est 'champ' ("field" en anglais)

**l'enregistrement est une structure de données hétérogène composée de champs**

Mais une telle structure (ici la variable *personne*) est de trop 'haut niveau' pour être exploitable directement et globalement – comme une variable scalaire; **on doit donc oublier** définitivement espérer écrire

- une affectation interne globale (en utilisant par exemple les accolades comme délimiteur de liste de composants et en respectant scrupuleusement l'ordre et le type des champs)

~~*personne* ← ( "Dupont", "Jean", 1950, 10, 21, "place H. Berger, 5a", 1300, "Wavre" )~~

- une affectation externe globale : ~~*lire(personne)*~~
- un affichage global du contenu : ~~*écrire(personne)*~~

**seuls les champs scalaires d'un enregistrement peuvent faire l'objet d'actions élémentaires telles que l'affectation et les opérations d'entrée/sortie**

<sup>55</sup> en Pascal : *record ... end*, en C : { ... }

<sup>56</sup> scalaires, dans un premier temps, mais on va rapidement dépasser cette approche étroite

b) L'OPÉRATEUR D'ACCÈS "POINT" (.)

Une structure d'Enregistrement n'étant pas utilisable globalement, il faut pouvoir manipuler les champs constitutifs; ici, le problème à régler, c'est l'accès à ces champs, ce qui pose un obstacle nouveau :

deux variables scalaires ne peuvent pas porter le même nom; mais si l'on déclare les deux structures ci-dessous, chacune contient des champs scalaires de même nom (ce qui est tout à fait autorisé) et bien entendu des instructions telles que `prénom ← ...`, `lire(prénom)` ou `écrire(prénom)` vont poser problème car elles sont ambiguës : le champ `prénom` de quelle structure utiliser ????

```
Variable étudiant : Enregistrement
    nom      : Texte
    prénom   : Texte
    ... # etc ...
finEnr
```

```
Variable enseignant : Enregistrement
    nom      : Texte
    prénom   : Texte
    ... # etc ...
finEnr
```

- chaque champ d'un enregistrement n'est accessible que par son nom 'complet' depuis l'extérieur (c-à-d depuis le nom générique), vers l'intérieur de la structure (c-à-d jusqu'au champ lui-même), en spécifiant les différents noms rencontrés le long du 'chemin' et en les séparant par un point (.) :

```
personne.prénom, personne.annéeNaissance, personne.codePostal ...
```

- ce . séparateur est en réalité un opérateur d'accès (en abrégé un accesseur) dont le rôle est d'opérer une résolution de nom non ambiguë (c-à-d. en programmation concrète par un langage : déterminer l'adresse-mémoire du champ)
- à partir de là, on se retrouve en terrain connu, puisque ces champs sont des variables scalaires à part entière et sont donc utilisables comme toute variable :

```
lire(personne.nom)
personne.annéeNaissance ← 1975
écrire(personne.codePostal)
```

**les champs d'un enregistrement ne sont accessibles que par leur nom complet, par l'intermédiaire de l'accesseur . appliqué au nom générique de la structure**  
**s'ils sont scalaires, ces champs peuvent faire l'objet d'actions élémentaires telles que l'affectation et les opérations d'entrée/sortie**

c) DÉCLARATIONS

La déclaration de principe d'une structure d'Enregistrement comporte

- ° un nom générique unique, suivi de ; ,
- ° suivi d'un marqueur de début de structure (Enregistrement),
- ° suivie d'une déclaration des différents champs constitutifs, (pour le moment) sous forme de déclarations de variables scalaires
- ° suivie d'un marqueur de fin de structure (finEnr)

```
... nomStructure : Enregistrement
    nomChamp1 : typeDuChamp1
    nomChamp2 : typeDuChamp2
    ...# etc ...
finEnr
```

d) LE MOT ENREGISTREMENT N'EST PAS UN IDENTIFICATEUR DE TYPE !

Pour être tout à fait rigoureux, l'enregistrement n'est pas à proprement parler un type (au sens des types scalaires rencontrés précédemment : Entier, Réel, Caractère, Logique, Texte ...) mais plutôt un outil 'générateur' (ou 'constructeur') de type (on parle aussi de méta-type); en effet,

- d'abord il n'est pas possible (car dépourvu de sens) de déclarer une variable comme étant simplement de "type Enregistrement" (point-barre !) :

```
# ceci ne veut rien dire : pas de structure, pas de contenu !
... client : Enregistrement

# ... mais ceci a du sens : un contenu déclaré en termes de composants
... client : Enregistrement
    numéroClient      : Entier
    nom                : Texte
    adresse            : Texte
    numéroCompte       : Texte
finEnr
```

- ensuite, nous l'avons déjà dit, la structure (via son nom générique) ne peut pas faire l'objet d'actions élémentaires<sup>57</sup>

```
# pas d'affectation globale ni externe, ni interne
lire(client)
client ← (345, "Lambert", "rue du Programme", "000-2345256-63")

# pas d'affichage global
écrire(client)
```

mais il est indispensable de passer par les champs, parce que les outils lire(), ←, écrire() ne savent manipuler que des scalaires :

```
lire(client.nom)
client.numéroCompte ← "000-2345256-63"
écrire(client.adresse)
```

Ceci nous amène évidemment à reconsidérer une fois encore la notion de valeur :

- ° pour les variables scalaires ordinaires pas de problème : leur valeur est le contenu de la 'boîte-mémoire' qui leur est allouée
- ° pour les structures, c'est un peu plus compliqué : on pourrait affirmer que la variable `client` ci-dessus possède une (et une seule) "valeur" à un moment déterminé : c'est simplement l'ensemble des valeurs prises par ses champs constitutifs, mais les instructions d'action élémentaires ne savent pas manipuler cette "valeur" de manière globale
- ° pour pouvoir disposer de fonctionnalités vraiment intéressantes sur les structures, et en particulier de les manipuler comme des valeurs à part entière, il va à présent falloir reconsidérer la notion de type

<sup>57</sup> les langages (Pascal, C, ...) offrent aujourd'hui la possibilité de déclarer des 'structures initialisées'; on y reviendra au second semestre en Organisation et Structure de Données et en Langage C

e) LES TYPES ABSTRAITS (ADT & API), OUTILS D'ABSTRACTION

Un premier pas vers la compréhension de la notion de 'valeur structurée typée' consiste à considérer la structure d'enregistrement en tant qu'outil d'abstraction au niveau des données :

- dans la vie courante, qu'elle soit privée ou professionnelle, nous sommes confrontés bien plus souvent qu'il n'y paraît à des "données structurées" :
  - pour une entreprise commerciale, les concepts de client, de fournisseur, de produit, de commande sont synonymes de structures (chacun comporte un ensemble de différentes informations, chacune décrivant une 'facette' particulière de la 'chose' représentée) ...
  - dans un établissement scolaire, il en va de même avec les notions d'étudiant, d'enseignant et de cours ...
  - un programme de télévision est une 'valeur structurée' (quel jour, quelle heure, quelle chaîne, quelle émission, quelle durée ...); il en va de même pour un module horaire dans une école (quelle classe/groupe, quel jour, quelle période, dans quel local, avec quel prof, pour quelle matière ...)
  - nous disposons de comptes bancaires, de cartes diverses (bancaires, fidélité, clubs ...) possédant chacune une structure spécifique (numéro, échéance, mot de passe, etc ...)
- bien que construites avec les mêmes outils (un nom générique, un nom et un type pour chaque champ) il serait faux de dire que toutes ces structures sont "identiques ou compatibles" ! cela a-t-il du sens - et si oui lequel - de comparer un client et un produit ? un étudiant et un cours ? une carte bancaire et un abonnement de train ?

pour en revenir aux variables scalaires élémentaires, on ne peut pas effectuer des opérations (telles que comparer ou additionner, p.ex.) entre des (valeurs) Entier et Texte, entre un Réel et un Logique ! etc ... pourquoi ? parce qu'ils n'ont pas le même type, pas la même nature !<sup>58</sup>

on va faire de même avec les structures de données; pour leur donner du sens (en termes de valeurs et d'opérations possibles) on va en faire de nouveaux types, c-à-d on va leur donner une nature précise !

- Pour les différencier des "types scalaires prédéfinis", on va qualifier cette démarche de création de "types-utilisateurs" ou mieux encore de "types de données abstraits" ou **ADT** (pour une fois, nous conserverons pour l'acronyme la terminologie anglophone Abstract Data Type qui est devenue un standard de fait, malgré les nombreuses tentatives de la littérature française d'utiliser TAD)
- pourquoi abstraits ? d'une part parce qu'ils sortent tout droit de l'imagination humaine (d'une problématique liée à la nécessité de représenter, de 'modéliser' le monde réel au mieux), d'autre part, parce qu'ils ne sont livrés avec aucun opérateur (quels opérateurs en effet permettraient de manipuler des données de type cours, de type horaire, de type abonnement ?)

**un type abstrait de données (ADT) est un type construit sur une structure de données; une variable d'un tel type possède une valeur à part entière**

- pour concrétiser ses nouveaux types de données, l'informaticien va devoir fabriquer des opérateurs adaptés; avec les seuls outils à sa disposition que sont les procédures et les fonctions (paramétrées), il va rédiger des **API** (Application Program Interface)

**les procédures et les fonctions sont les seuls outils procéduraux qui – grâce aux paramètres typés – peuvent manipuler des variables structurées typées comme des valeurs; elles constitueront l'API spécifiques de l'ADT**

<sup>58</sup> on remarquera cependant que si l'on a des variables scalaires réelles *poids* et *taille*, qu'est-ce qui nous empêche de les comparer ou de les additionner ? ... et quel est alors le sens d'une telle opération ? ... et de son résultat ?

Il est grand temps de montrer tout cela concrètement :

ainsi, pour une application de gestion d'établissement scolaire, plutôt que de déclarer séparément les deux (variables-)structures distinctes suivantes :

```
Variable etu1 : Enregistrement
    matricule      : Texte
    nom            : Texte
    prénom        : Texte
    annéeNaissance : Entier
    moisNaissance  : Entier
    jourNaissance  : Entier
    adresse        : Texte
    codePostal     : Entier
    ville          : Texte
finEnr
```

```
Variable etu2 : Enregistrement
    matricule      : Texte
    nom            : Texte
    prénom        : Texte
    annéeNaissance : Entier
    moisNaissance  : Entier
    jourNaissance  : Entier
    adresse        : Texte
    codePostal     : Entier
    ville          : Texte
finEnr
```

on préférera déclarer (et donc créer) un nouveau type abstrait : un ADT

```
Type TEtudiant = Enregistrement
    matricule      : Texte
    nom            : Texte
    prénom        : Texte
    annéeNaissance : Entier
    moisNaissance  : Entier
    jourNaissance  : Entier
    adresse        : Texte
    codePostal     : Entier
    ville          : Texte
finEnr
```

de manière à créer facilement plusieurs variables de ce type (comme on a pu le faire avec les types scalaires prédéfinis) :

```
Variable etu1, etu2 : TEtudiant
```

Qu'est-ce qu'on y gagne ? Tout ! Car désormais, puisqu'elles se réclament d'un type existant les variables `etu1` et `etu2` possèdent une 'vraie' valeur (puisque cette notion est intimement liée au type : seules les données typées possèdent une valeur)

- On récupère d'un seul coup l'affectation (mettre une valeur dans une variable); on pourra donc écrire :  

```
etu2 ← etu1           # puisque etu1 possède une valeur typée et que
                       etu1 et etu2 possèdent le même type !
```
- Les procédures et fonctions paramétrées sont les outils avec lesquels le programmeur va définir les 'opérateurs de haut niveau' pour le nouveau type; par exemple ci-dessous, nous allons écrire une nouvelle 'affectation externe' pour des variables de type `TEtudiant` (on part de l'idée que si on peut écrire `etu2 ← etu1`, alors `etu1` peut être remplacé par une fonction ...

La rédaction de la fonction ci-dessous est une écriture de principe, nous avons volontairement omis d'y placer les invites nécessaires et les validations indispensables (date, code postal, p.ex.)

```
Fonction lireEtu() : TEtudiant      # fournit une 'valeur' de type étudiant
Variable etu : TEtudiant           # variable locale de type étudiant
Début
    lire(etu.matricule)             # lecture des différents champs scalaires
    lire(etu.nom)
    lire(etu.prénom)
    lire(etu.annéeNaissance)
    lire(etu.moisNaissance)
    lire(etu.jourNaissance)
    lire(etu.adresse)
    lire(etu.codePostal)
    lire(etu.ville)
    renvoie(etu)                   # fournit la valeur au demandeur
Fin
```

mettons tout en place dans un algorithme ...

```
Algorithme Etudiants :
Type TEtudiant = Enregistrement    # ADT étudiant
    matricule      : Texte
    nom            : Texte
    prénom         : Texte
    annéeNaissance : Entier
    moisNaissance  : Entier
    jourNaissance  : Entier
    adresse        : Texte
    codePostal     : Entier
    ville          : Texte
finEnr
Variable etu1, etu2, etu3 : TEtudiant # variables globales de l'algo
Fonction lireEtu() : TEtudiant      # API : fournit une 'valeur' étudiant
Variable etu : TEtudiant           # variable locale de type étudiant
Début
    lire(etu.matricule)             # lecture des différents champs scalaires
    lire(etu.nom)
    lire(etu.prénom)
    lire(etu.annéeNaissance)
    lire(etu.moisNaissance)
    lire(etu.jourNaissance)
    lire(etu.adresse)
    lire(etu.codePostal)
    lire(etu.ville)
    renvoie(etu)                   # fournit la valeur au demandeur
Fin
Début
...
    etu1 ← lireEtu()               # affectation externe globale généralisée
    etu2 ← lireEtu()
    etu3 ← etu2                    # affectation interne globale généralisée
...
Fin
```

dans le même ordre d'idée, il suffit de rédiger une procédure écrireEtu(e : TEtudiant) pour disposer d'une procédure globale d'affichage ...

La déclaration d'un nouveau type (utilisateur ou ADT) obéit à la syntaxe suivante :

... *nomType* = définition du type

f) STRUCTURES ET SOUS-STRUCTURES ...

La structure d'enregistrement n'est pas limitée à un seul niveau : en fonction de la structure de données nécessaire pour abstraire la partie données d'un algorithme, on peut imbriquer des structures, par exemple :

*pour modéliser le type personne, on peut ainsi déclarer ...*

```

Type TPersonne = Enregistrement
  nom      : Texte
  prenom   : Texte
  naissance : Enregistrement
    jour : Entier
    mois : Entier
    année : Entier
  finEnr
  adresse : Enregistrement
    rue      : Texte
    codePostal : Entier
    ville    : Texte
  finEnr
finEnr
Variable etu, prof : TPersonne

```

*... mais il est toujours préférable (pour la réutilisabilité potentielle) de jouer 'à fond' la carte du typage abstrait*

```

Type TDate = Enregistrement
  jour : Entier
  mois : Entier
  année : Entier
finEnr

Type TAdresse = Enregistrement
  rue      : Texte
  codePostal : Entier
  ville    : Texte
finEnr

Type TPersonne = Enregistrement
  nom      : Texte
  prenom   : Texte
  naissance : TDate
  adresse  : TAdresse
finEnr
finEnr
Variable etu, prof : TPersonne

```

Remarques importantes :

- attention : les identificateurs de type ne sont utilisés que dans la déclaration (à gauche du symbole = lors de leur 'baptême' ou à droite d'un symbole : lorsqu'ils précisent le type d'un (sous-)composant), lors de l'utilisation c'est toujours le nom concret de la structure qui est utilisé

ainsi, qu'il s'agisse de la déclaration de gauche ou de celle de droite ci-dessus, l'accès aux champs des variables suit la même logique de 'chemin' au moyen de l'accessueur .

```

etu.nom, etu.naissance.mois, etu.adresse.codePostal, ...
prof.naissance.année, prof.adresse.rue, ...

```

alors qu'il est incorrect (à droite) d'écrire

~~etu.TDate.mois, etu.TAdresse.codePostal, ...~~

- même si elle paraît plus lourde à première vue, on ne peut que conseiller la version de droite; en multipliant les ADT, on renforce l'idée de 'valeurs' typées (on y voit apparaître comme telles les dates, les adresses ...) confirmée par le fait qu'on écrira des 'petites' API spécialisées (une fonction renvoyant une date validée, une autre renvoyant une date formatée<sup>59</sup>, p.ex.; une fois de plus, ci-dessous, ni les invites ni les validations n'ont été écrites)

```

Fonction lireDate() : TDate      # API : fournit une 'valeur' de type date
Variable d : TDate               # variable locale de type date
Début
  lire(d.jour)                   # lecture des différents champs scalaires
  lire(d.mois)
  lire(d.année)
  renvoie(d)                     # fournit la date au demandeur
Fin
...
etu.naissance ← lireDate()

```

<sup>59</sup> voir les exercices proposés en fin de chapitre ...



g) SUCRE SYNTAXIQUE

Lorsque la structure est complexe et à plusieurs niveaux (mais on peut déjà faire cette réflexion avec un seul niveau), on peut considérer comme 'lourde' l'obligation de spécifier le chemin complet de chaque champ de la structure lors de son utilisation, p.ex. :

```

Type TDate = Enregistrement
  jour   : Entier
  mois   : Entier
  année  : Entier
finEnr

Type TAdresse = Enregistrement
  rue      : Texte
  codePostal : Entier
  ville    : Texte
finEnr

Type TPersonne = Enregistrement
  nom       : Texte
  prénom    : Texte
  naissance : TDate
  adresse   : TAdresse
finEnr

Variable etu, prof : TPersonne
...
Procédure encoderEtudiant :
Début
  lire(etu.nom)
  lire(etu.prenom)
  lire(etu.naissance.jour)
  lire(etu.naissance.mois)
  lire(etu.naissance.année)
  lire(etu.adresse.rue)
  lire(etu.adresse.codePostal)
  lire(etu.adresse.ville)
Fin
...
```

c'est la raison pour laquelle il existe (parfois, dans des langages qui proposent la structure d'enregistrement, mais pas toujours<sup>60</sup>) un outil algorithmique à caractère structurel : le préfixage ('raccourci de chemin')

```

Procédure encoderEtudiant :
Début
  Avec etu :                # préfixera ce qui suit avec etu.
    lire(nom)                # (= etu.nom)
    lire(prénom)
    Avec naissance :        # préfixera la suite avec naissance. (donc avec etu.naissance.)
      lire(jour)              # (= etu.naissance.jour)
      lire(mois)
      lire(année) finAvec
    Avec adresse :          # préfixera la suite avec adresse. (donc avec etu.adresse.)
      lire(rue)               # (= etu.adresse.rue)
      lire(codePostal)
      lire(ville) finAvec
  finAvec
Fin
```

<sup>60</sup> c'est le cas en Pascal (avec la clause **with**), mais pas en C par contre ...

Une fois encore, on peut souhaiter (conseiller de) décomposer en unités plus simples, pour la réutilisabilité éventuelle; on écrirait donc plutôt (en utilisant ici les variables globales, plutôt que de passer par des fonctions et/ou procédures ... on tournera donc vite la page ... ) :

```

Type TDate = Enregistrement
  jour   : Entier
  mois   : Entier
  année  : Entier
finEnr

Type TAdresse = Enregistrement
  rue           : Texte
  codePostal    : Entier
  ville         : Texte
finEnr

Type TPersonne = Enregistrement
  nom           : Texte
  prénom       : Texte
  naissance     : TDate
  adresse       : TAdresse
finEnr

Variable etu, prof : TPersonne

Procédure encoderEtuDate :
Début
  Avec etu.naissance :   # prefixera tout ce qui suit avec etu.naissance.
    lire(jour)
    lire(mois)
    lire(année)
  finAvec
Fin

Procédure encoderEtuAdresse :
Début
  Avec etu.adresse :     # prefixera tout ce qui suit avec etu.adresse.
    lire(rue)
    lire(codePostal)
    lire(ville)
  finAvec
Fin

Procédure encoderEtudiant :
Début
  Avec etu :              # prefixera tout ce qui suit avec etu.
    lire(nom)
    lire(prénom)
    encoderEtuDate
    encoderEtuAdresse
  finAvec
Fin
...
encoderEtudiant

```

h) API : PROCÉDURES OU FONCTIONS ?

Si l'on regarde attentivement l'exemple de la page précédente et que l'on imagine les déclarations suivantes

```
Type TDate = Enregistrement
  jour   : Entier
  mois   : Entier
  année  : Entier
finEnr

Type TEtudiant = Enregistrement
  nom      : Texte
  prénom   : Texte
  naissance : TDate
  ...
  inscription : TDate
  ...
finEnr

Variable etu1, etu2 : TEtudiant
...
```

on se rend compte que c'en est fini des procédures et des fonctions qui manipulent directement les variables globales et que désormais, on aura recours à des versions plus abstraites de ces outils, via le passage de paramètres

reste à déterminer – par exemple ici pour l'encodage des dates – si l'on aura recours plutôt à une fonction ou plutôt à une procédure

- la fonction renforce l'idée de valeur (une structure typée possède une valeur) et privilégie l'affectation interne

```
Fonction lireDate() : TDate      # API : fournit une 'valeur' de type date
Variable d : TDate              # variable locale de type date
Début
  lire(d.jour)                  # lecture des différents champs scalaires
  lire(d.mois)
  lire(d.année)
  renvoie(d)                  # fournit la date au demandeur
Fin
...
etu1.naissance ← lireDate()
etu2.inscription ← lireDate()
```

- la procédure généralisera plutôt l'idée d'affectation externe; sa rédaction est quelque peu plus légère MAIS on devra lui transmettre par référence la structure de données à modifier

```
Procédure lireDate(ref d : TDate) :      # paramètre par référence
Début
  lire(d.jour)                  # lecture des différents champs scalaires
  lire(d.mois)
  lire(d.année)
Fin
...
lireDate(etu1.naissance)
lireDate(etu2.inscription)
```

La fonction semble mieux convenir, mais elle possède une limite importante : elle ne sait transmettre qu'une seule valeur; on rencontrera des situations où, parce que plusieurs valeurs sont modifiées 'ensemble', le recours à une procédure peut donc sembler (à première vue) la seule solution possible ... (patience)

i) CONCRÈTEMENTFortran

A l'origine (1953), c'est un langage conçu principalement pour les scientifiques ... manifestement ceux-ci n'ont jamais émis le besoin ou le désir de disposer d'une structure de données inhomogène de type Enregistrement ... de sorte qu'une première forme de ce type de structure n'y apparaît que dans les versions des années 1990 ... et elle sera remplacée ensuite par l'orientation Objet du langage.

petit exemple : on commence par définir un 'type abstrait' (ici article) composés de 3 champs scalaires

```
STRUCTURE /ARTICLE/
  INTEGER REFERENCE
  CHARACTER(LEN=200) DESCRIPTION
  REAL PRIX
END STRUCTURE
```

puis on déclare des variables de ce 'type'

```
RECORD /ARTICLE/ LIVRE, CAHIER
```

et dans le code, on utilise l'accesseur . (point) pour référencer les champs dans les variables

```
LIVRE.REFERENCE = 2345
LIVRE.DESCRPTION = "COURS DE FORTRAN"
LIVRE.PRIX = 33.5
```

Cobol

Langage orienté gestion, il possède en natif la structure d'enregistrement, qui se déclare sous forme hiérarchique, à l'aide niveaux numérotés (01 pour la 'racine') avec toutes sortes de possibilités pour pré-initialiser les champs, y associer des états binaires, etc ...

```
01 ETUDIANT.
  02 MATRICULE          PIC X(8).
  02 SEXE               PIC X.
  02 IDENTITE.
    03 PRENOM           PIC X(20).
    03 INITIALE         PIC X.
    03 NOM              PIC X(35).
  02 NAISSANCE.
    03 DATENAISS.
      04 JOUR           PIC 99.
      04 MOIS          PIC 99.
      04 ANNEE          PIC 9(4).
    03 VILLENAISS       PIC X(30)

01 PROFESSEUR.
  02 REFERENCE          PIC X(8).
  02 SEXE               PIC X.
```

Les champs sont accessibles directement sans accesseur, quelle que soit leur 'position' dans la structure, pour autant qu'il n'y ait pas d'ambiguïté (que ce nom soit unique)

```
MOVE 'HE201212' TO MATRICULE.
```

s'il y a ambiguïté, l'accesseur est le mot OF

```
MOVE 'M' TO SEXE OF PROFESSEUR.
```

Malheureusement, Cobol n'est pas un langage très 'intello' et il n'y a quasi pas de place pour l'abstraction : pas de type abstrait (donc pas d'ADT), pas de possibilité de fonction avec passage de paramètres (donc pas d'API)

## Pascal

Tout en Pascal favorise l'abstraction et l'expressivité : structures de données (record), types abstraits (type), fonctions (function) et procédures (procedure) paramétrées : le couple ADT + API est offert au programmeur

côté ADT, les types abstraits se déclarent d'abord (juste après les constantes) :

```
type TDate = record
  jour : integer;
  mois : integer;
  annee : integer
end;

type TPersonne = record
  nom : string;
  prenom : string;
  naissance : TDate
end;
```

les variables structurées font référence à ces types-utilisateurs :

```
var etu, prof : TPersonne;
```

puis l'API est construite au moyen de fonctions (pour généraliser l'affectation interne) ...

```
function fLireDate() : TDate;
var d : TDate
begin
  with d do begin
    read(jour);
    read(mois);
    read(annee)
  end;
  fLireDate ← d
end;

function fLirePersonne() : TPersonne;
var p : TPersonne;
begin
  with p do begin
    read(nom);
    read(prénom);
    naissance ← fLireDate();
  end;
  fLirePersonne ← p
end;
```

... ou de procédures (pour généraliser l'affectation externe) :

```
procedure pLireDate(var d : TDate);
begin
  with d do begin
    read(jour);
    read(mois);
    read(annee)
  end
end;

procedure pLirePersonne(var p : TPersonne);
begin
  with p do begin
    read(nom);
    read(prénom);
    pLireDate(naissance)
  end;
end;
```

et dans la séquence principale (programme), les variables peuvent indifféremment être encodées via affectation interne généralisée (par invocation de fonction)

```
begin
  ...
  etu ← fLirePersonne;
  ...
end.
```

ou via affectation externe généralisée (procédure avec passage de paramètre par adress/référence)

```
begin
  ...
  pLirePersonne(prof);
  ...
end.
```

## C

comme Pascal, C propose – mais à sa manière - tous les outils d'abstraction et de réutilisation nécessaires à une programmation de haut niveau

typage abstrait (typedef) d'enregistrements (struct) :

```
typedef struct {
  short jour;
  short mois;
  int annee;
} TDate;

typedef struct {
  char nom[35];
  char prenom[20];
  TDate naissance;
} TPersonne;
```

'vraies' fonctions

```
TDate fLireDate(void) {
  TDate d;
  scanf("%d", &d.jour);
  scanf("%d", &d.mois);
  scanf("%d", &d.annee);
  return(d);
}
```

ou 'vraies' procédures avec passage de paramètres par adresse

```
void pLirePersonne(TPersonne* p) {
  scanf("%s", p.nom);
  scanf("%s", p.prenom);
  *p.naissance = fLireDate();
}
```

utilisées dans le programme (fonction) principal main

```
int main() {
  TPersonne etu, prof;    // déclaration variables
  ...
  pLirePersonne(&etu);    // via procédure
  prof = fLirePersonne(); // via fonction

  return 0;
}
```

j) EXERCICES

- 1) Étant donné le type abstrait TDate défini comme suit :

```
Type TDate = Enregistrement
  jour   : Entier
  mois   : Entier
  année  : Entier
finEnr
```

récrivez une fonction de saisie (complète, avec invite et validation) dont la signature est

```
Fonction lireDate() : TDate
```

pour la validation, cette fonction devra obligatoirement invoquer une autre fonction (à rédiger également) dont la signature est

```
Fonction dateValide(d : TDate) : Logique
```

- 2) Pour informatiser la gestion d'un établissement scolaire, on vous demande d'imaginer les structures de données permettant de modéliser les étudiants et les enseignants; vous vous rendez vite compte que ces personnes ont beaucoup de points communs : un nom, un prénom, une date de naissance, un domicile (c-à-d une adresse), un téléphone fixe et un téléphone mobile, une adresse-mail. Cependant, les étudiants possèdent en propre un matricule et une date d'inscription, alors que les enseignants ont une date d'entrée en fonction, une ancienneté (en années), un bureau et un n° de téléphone intérieur.

A l'aide de la structure d'Enregistrement, quel(s) type(s) abstrait(s) allez-vous définir pour modéliser de la manière la plus souple les structures d'étudiant et d'enseignant ?

- 3) Rédigez un ADT permettant de représenter les informations d'une référence bibliographique : le titre du livre, le nom de l'auteur, le nom de l'éditeur, l'année de publication et le nombre de pages.

Ensuite, rédigez une API pour pouvoir encoder et afficher facilement une telle référence bibliographique

- 4) Vous participez à l'analyse d'un logiciel de gestion pour une société commerciale; vous découvrez les déclarations suivantes :

```
Variable Client : Enregistrement
  noCli   : Entier           # numéro de client
  nomCli  : Entier
  prénomCli : Entier
  rueAdrCli : Texte         # adresse du client
  codeAdrCli : Entier
  villeAdrCli : Texte
  télCli  : Texte
  faxCli  : Texte
  gsmCli  : Texte
finEnr
```

```
Variable Fournisseur : Enregistrement
  noFourn  : Entier           # numéro de fournisseur
  nomFourn  : Entier
  prénomFourn : Entier
  rueAdrFourn : Texte        # adresse du fournisseur
  noAdrFourn : Texte
  villeAdrFourn : Texte
  télFourn  : Texte
  faxFourn  : Texte
  gsmFourn  : Texte
  noTva     : Texte
finEnr
```

```
Variable Commande : Enregistrement
noComm : Entier           # numéro de bon de commande
noFourn : Entier          # numéro de fournisseur
jourComm : Entier         # date de commande
moisComm : Entier
annéeComm : Entier
rueCommLivr : Texte      # adresse livraison
codeCommLivr : Entier
villeCommLivr : Texte
jourLivr : Entier        # date de livraison
moisLivr : Entier
annéeLivr : Entier
rueCommFact : Texte      # adresse facturation
codeCommFact : Entier
villeCommFact : Texte
jourFact : Entier        # date de facturation
moisFact : Entier
annéeFact : Entier
montantFact : Réel       # montant de la facture
jourPai : Entier         # date de paiement
moisPai : Entier
annéePai : Entier
montantPai : Réel       # montant du paiement
finEnr
```

... et on vous précise que la variable Commande sert aussi bien à mémoriser une commande passée à l'entreprise par un client (alors le champ noFourn est vide) qu'une commande passée par l'entreprise à un fournisseur (dans ce cas, c'est le champ noCli qui est vide)

Que pouvez-vous proposer pour rationaliser tout cela ?



### 5.3. LA STRUCTURE DE TABLEAU

#### a) PREMIER CONTACT

Si l'on reprend une problématique évoquée à l'entame de ce chapitre (l'encodage, le classement et l'affichage de cotes), on est cette fois face à un cas où les données sont de même type (toutes les cotes sont de type Réel), mais où l'on sent bien qu'il manque 'quelque chose' du côté des données, qui serait – toutes proportions gardées – l'équivalent de la boucle côté algorithmique afin de pouvoir 'répéter' plusieurs fois une même variable (mais sous un seul et même nom, contrairement à ce qui a été fait, où il a fallu une variable explicite pour chaque donnée)

C'est exactement ce que va nous offrir la structure de Tableau. Elle est présente dès les tout premiers langages de programmation (dès le Fortran, en 1953), tout simplement parce que la mémoire est constituée d'éléments adjacents numérotés et que la fonctionnalité d'adressage d'un bloc de mémoire par 'base et déplacement' – via les registres d'index<sup>61</sup> – a toujours existé dans le jeu d'instruction des processeurs (donc en 'langage machine', puis en assembleur ...)

Le Tableau (array en anglais) est une structure de données composée d'un ensemble d'éléments (appelés parfois cases, cellules, ...).

Dans les langages compilés 'classiques' statiquement typés, (depuis Fortran, en passant par Cobol, Pascal, C, Java ...) tous les éléments d'un tableau doivent être du même type, on dit alors que la structure est homogène.<sup>62</sup>

Cette caractéristique essentielle va se retrouver dans la déclaration d'une telle structure, par exemple

```
Variable tablCotes : Tableau ... de Réel
Variable tablPrésences : Tableau ... de Logique
Variable tablNoms : Tableau ... de Texte
```

On voit apparaître (provisoirement) :

- un nom symbolique (un identificateur donné par le programmeur) pour l'ensemble de la structure (ici **tablCotes**, ..., **tablNoms**); ce nom sera ici également qualifié de nom générique (on y reviendra)
- la spécification du type de chaque élément de cette structure (**Réel**, ..., **Texte**); on pourra utiliser la qualification de type générique

**le tableau est une structure de données homogène composée de cellules de même type**

Mais ici également, une telle structure (p.ex. la variable **tablCotes**) est de trop 'haut niveau' pour être exploitable directement et globalement, comme une variable scalaire; **on doit donc oublier** définitivement espérer écrire

- une affectation interne globale (en utilisant par exemple les accolades comme délimiteur de liste et en respectant scrupuleusement le type des cellules)

~~**tablCotes** ← { 12.0, 11.5, 9.3, 17.0, 18.5, 5.5, 13.5 }~~

- une affectation externe globale : ~~**lire(tablCotes)**~~
- un affichage global du contenu : ~~**écrire(tablCotes)**~~

**seules les cellules scalaires d'un tableau peuvent faire l'objet d'actions élémentaires telles que l'affectation et les opérations d'entrée/sortie**

<sup>61</sup> voir le chapitre consacré à "la Mémoire et sa Gestion" dans le cours d'Initiation à l'informatique

<sup>62</sup> Dans les langages récents (comme Python, Ruby, etc), interprétés, et offrant le typage dynamique, cette restriction a disparu (le présent texte restera cependant très 'classique' en se limitant aux seuls tableaux homogènes).

Se limiter à une déclaration spécifiant un nom générique et un type générique de composants est insuffisant : il faut en effet préciser également le nombre de composants que la structure va comporter (rappelons que les compilateurs des langages statiques doivent réserver de la place en mémoire, d'où la nécessité de cette information "nombre et type de composants")

Complétons donc la syntaxe générale de la déclaration d'un tableau, par exemple

```
Variable tablCotes : Tableau[100] de Réel
Variable tablPrésences : Tableau[25] de Logique
Variable tablNoms : Tableau[30] de Texte
```

On voit donc apparaître (explicitement et finalement) :

- un nom générique pour la structure (comme précédemment)
- la spécification type générique des composants (comme précédemment)
- le nombre de composants placé entre crochets derrière le mot Tableau

## b) L'OPÉRATEUR D'ACCÈS [ ]

Comme pour la structure Enregistrement, seules les composants (cellules) d'un Tableau peuvent faire l'objet d'actions élémentaires, mais – contrairement aux champs d'un Enregistrement – elles n'ont pas reçu chacune un nom lors de la déclaration, d'où la question : quel est le nom des cellules ?

au sein d'un Tableau, chaque cellule est repérée par sa position, appelée indice, la première possède l'indice 1 et la dernière un indice égal au nombre d'éléments déclarés

le nom de toute cellule d'un Tableau est obtenu à partir du nom générique spécifié lors de la déclaration et de l'indice de cette cellule via l'opérateur d'accès (l'accesseur) [ ]

contrainte : la valeur de l'indice doit obligatoirement être une valeur entière comprise dans l'intervalle défini implicitement par le nombre d'éléments du Tableau, c-à-d [1 .. nbElem]

1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
														...

Avec la déclaration : Variable **tablCotes** : **Tableau**[100] de **Réel**

la première cellule s'appelle **tablCotes**[1]

la dixième s'appelle **tablCotes**[10]

et la dernière s'appelle **tablCotes**[100]

On s'empressera dès à présent d'appliquer un des grands principes de programmation étudiés jusqu'ici : l'usage des constantes symboliques (leurs avantages ne sont plus à expliquer ...); on préférera donc écrire :

```
Constante nbElem = 50
Variable tablCotes : Tableau[nbElem] de Réel
```

de la sorte, on pourra appeler (aussi) la dernière cellule **tablCotes**[**nbElem**]

En programmation (statique), les tableaux homogènes sont considérés comme des structures de données à « accès direct » : le temps d'accès à un élément par son indice est en effet constant, quel que soit l'élément désiré. Cela s'explique par trois raisons :

- les éléments du tableau sont contigus dans l'espace mémoire
- via le type générique, on connaît la taille d'un élément (et elle est identique pour tous les éléments)
- l'indice n'est qu'un déplacement par rapport au début du tableau : l'adresse de n'importe quel élément en mémoire peut donc être calculée très facilement.

c) DÉCLARATIONS ET NOTATIONS

Classiquement, dans la plupart des langages de programmation, on déclare (statiquement, c.-à-d. que ce n'est pas modifiable ensuite, au cours de l'exécution) le type et le nombre d'éléments par une déclaration du genre :

**... nomTableau : Tableau[nombreÉléments] de typeDeComposant**

exemple : Variable `tablNoms` : Tableau[20] de Texte  
 Variable `tablCotes` : Tableau[20] de Réel

Il s'agit donc – comme annoncé précédemment – d'une déclaration générique : un\_seul\_et\_même\_nom pour l'ensemble de la structure (le tableau), un\_seul\_et\_même\_type pour tous les composants de la structure (les cellules).

De manière quasi universelle, les cellules individuelles sont nommées à partir du nom générique du tableau et de leur position (indice) à l'aide de l'accesseur [ ] :

exemple : `tablNoms[4] ← "PIERRARD"` # dans la quatrième cellule, mettre ...  
`tablCotes[10] ← 13.5` # dans la dixième cellule, mettre ...

... mais ceci pose un problème, celui de l'indice de départ : selon les langages, le premier élément possède l'indice 0 (C, Java, ...) <sup>63</sup>, pour d'autres l'indice 1 (Fortran, Cobol, Pascal) <sup>64</sup>

Pour rendre l'apprentissage de la structure de Tableau plus intuitif, nous prendrons ici comme seule et unique convention que la première cellule possède l'indice 1

d) LE MOT TABLEAU N'EST PAS UN IDENTIFICATEUR DE TYPE !

Faisons ici une remarque capitale semblable à celle énoncée dans le cas de la structure d'Enregistrement : le Tableau n'est pas à proprement parler un type (au sens des types scalaires rencontrés précédemment : Entier, Réel, Caractère, Logique ...) mais plutôt un outil 'générateur' (ou 'constructeur') de type (on parle aussi de meta-type)

- d'abord il n'est pas possible (car dépourvu de sens, du moins pour les langages compilés) de déclarer une variable comme étant simplement de "type Tableau" (point-barre !) :

# ceci ne veut rien dire : pas de structure, pas de contenu !  
~~Variable `tablCotes` : Tableau~~  
 # ceci a du sens : un contenu déclaré en termes de composants typés  
 Variable `tablCotes` : Tableau[20] de Réel

- ensuite, nous l'avons déjà dit, la structure (via son nom générique) ne peut pas faire l'objet d'actions élémentaires

~~# pas d'affectation globale :  
 # ... ni externe  
 lire(`tablCotes`)  
 # ... ni interne  
`tablCotes` ← { 12.5, 6.3, 10.9, 11.2, 13.0, ...}  
 # pas d'affichage global  
 écrire(`tablCotes`)~~

<sup>63</sup> ce qui se justifie du point de vue de l'implémentation : si l'indice commence à 0, l'adresse de l'élément d'indice  $i$  est : adresse de début +  $i$  \* longueur d'un élément, alors que si l'indice commence à 1, l'adresse de l'élément d'indice  $i$  est : adresse de début +  $(i-1)$  \* longueur d'un élément (on gagne une soustraction lors du calcul d'adresse avec la première méthode)

<sup>64</sup> en Pascal, les possibilités offertes au programmeur lui permettent d'indicer avec tous les types de données scalaires énumérables (integer, char, boolean) et la déclaration du nombre d'éléments est indirecte à partir de la déclaration de l'intervalle au sein duquel l'indice pourra prendre ses valeurs; voir le cours de Pascal pour plus de détails...

mais il est indispensable de passer par les cellules, parce que les outils lire(), ←, écrire() ne savent manipuler que des scalaires (et que seules les cellules ont ce statut) :

```
lire(tablCotes[6])
tablCotes[2] ← 15.5
écrire(tablCotes[19])
```

Ceci nous amène évidemment à reconsidérer une fois de plus la notion de valeur :

- ° pour les variables scalaires ordinaires pas de problème : leur valeur est le contenu de la 'boîte-mémoire' qui leur est allouée
- ° pour les Tableaux comme pour les Enregistrements, c'est un peu plus compliqué : on peut affirmer que la variable `tablCotes` ci-dessus possède une (et une seule) "super-valeur" à un moment déterminé : c'est simplement l'ensemble des valeurs prises par ses cellules, mais les instructions d'action élémentaires ne savent pas manipuler cette "super-valeur" de manière globale
- ° pour pouvoir disposer de fonctionnalités vraiment intéressantes sur les Tableaux, il va à présent falloir considérer les ADT "Tableaux" ainsi que leurs API

On écrira donc

```
Type TtbCotes = Tableau[50] de Réel
```

de manière à créer facilement plusieurs variables de ce type (comme on a pu le faire avec les types scalaires prédéfinis, puis avec les Enregistrement typés) :

```
Variable tblCotes1TL, tblCotes1TM : TtbCotes
```

Mais patience, il y a tellement de choses à découvrir avant cela ...

e) ITÉRATEURS BASIQUES POUR UN TABLEAU À UNE DIMENSION

Restons quelque temps encore avec des structures de Tableau plus 'basiques' ... (non typées)

À partir des déclarations suivantes (non optimales du point de vue formel, cfr. plus loin) :

```
Variable tbl : Tableau[10] de Entier    # variable-tableau
      indice : Entier                  # indice de position de cellule
```

on va pouvoir – enfin - écrire une boucle qui va parcourir la structure, élément par élément, (p.ex. ci-dessous pour initialiser chaque case du tableau à 0) ...

puisque l'indice qui précise la position de la cellule est une 'valeur entière', toute valeur entière (littérale, constante symbolique, variable, fonction, et même toute expression à valeur entière) est admise pour autant qu'elle respecte la contrainte : *valeur entre 1 et nombre déclaré d'éléments*

!!!! avant de nous lancer dans cette aventure, une dernière remarque : les Tableaux sont des "tueurs" d'applications : il suffit au programmeur (débutant ou même aguerri) de mal maîtriser les indices pour déborder de la zone-mémoire allouée à la structure avec toutes les conséquences dramatiques que cela implique (depuis une application qui 'plante', ça c'est une bonne chose, à celle qui donne des résultats incorrects et inattendus, c'est franchement le pire cas de figure) !!!

Ces boucles, à présent, on va pouvoir les rédiger de diverses manières :

- avec les boucles tantQue et Répéter
- en initialisant la variable de parcours/positionnement (indice) à 1 (ci-dessous à gauche) ou à 0 (ci-dessous à droite)
- remarquer dans toutes ces différentes versions l'écriture spécifique de la condition de continuation/arrêt ainsi que la position de l'instruction d'incrémentatation indice  $\leftarrow$  indice + 1

```
Procédure initialiser1a :
Début
  indice ← 1
  tantQue non (indice > 10)
    tbl[indice] ← 0
    indice ← indice + 1
  finTQ
Fin
```

```
Procédure initialiser1b :
Début
  indice ← 0
  tantQue non (indice = 10)
    indice ← indice + 1
    tbl[indice] ← 0
  finTQ
Fin
```

```
Procédure initialiser2a :
Début
  indice ← 1
  Répéter
    tbl[indice] ← 0
    indice ← indice + 1
  Jusque indice > 10
Fin
```

```
Procédure initialiser2b :
Début
  indice ← 0
  Répéter
    indice ← indice + 1
    tbl[indice] ← 0
  Jusque indice = 10
Fin
```

comme a dit un grand 'savant' de l'informatique<sup>65</sup> *"quatre possibilités pour faire une même chose, c'est sans doute trois de trop !"*

on aurait tendance à privilégier les versions 1b et 2b, simplement parce que les conditions d'arrêt/continuation s'expriment de manière naturelle et directe en fonction de l'égalité au nombre d'éléments déclarés (indice = 10)

de toute façon, devoir écrire 5 lignes pour exécuter une seule instruction intéressante (tbl[indice] ← 0), on peut à juste titre estimer que c'est lourd !

<sup>65</sup> E. Codd, inventeur du Modèle Relationnel (socle des Bases de Données Relationnelles)

f) L'ITÉRATEUR "POUR"

Pour toutes ces raisons, c-à-d. éviter à chaque programmeur

- d'avoir à choisir un des quatre styles de boucle
- de se tromper dans la ligne d'initialisation, voire de l'oublier !
- de mal maîtriser la position de la ligne d'incrémentation de l'indice, voire de l'oublier (on remarquera que c'est la seule qui contribue à rendre vrai l'état : *on est arrivé au bout du tableau*)
- de se tromper dans l'expression de la condition d'arrêt/continuation (le danger d'aller *'une case trop loin'* ou *'une case pas assez loin'*)

la structure de Tableau est 'livrée' avec une boucle "fermée" appelée plus spécifiquement itérateur qui 'se charge de tout' ! sa syntaxe est la suivante :

```
Pour indice de valDébut à valFin par p Faire
    instruction/séquence
finPr
```

on se contente d'y spécifier

- le nom de la variable qui servira d'indice aux cellules
- les bornes (valeurs) inférieure et supérieure d'un intervalle autorisé pour les valeurs de cet indice
- ...et la 'boucle' se charge intégralement du reste : incrémentation de l'indice à chaque itération et gestion de l'arrêt correct de la boucle, les quatre boucles évoquées précédemment sont avantageusement (pour le programmeur) remplacées par celle-ci :

```
Procédure initialiser3 :
Début
    Pour indice de 1 à 10 Faire
        t[indice] ← 0
    finPr
Fin
```

- on n'insiste jamais assez, la boucle Pour est un 'vrai' tantQue, comportant donc et heureusement un principe de précaution (comme la version 1a que nous avons écrite à la page précédente), gérée (en interne) de la manière suivante :
  - la valeur valDébut est affectée à la variable indice
  - la condition `indice > valFin` est immédiatement évaluée (si elle est vraie, la boucle ne sera pas exécutée, situation que nous rencontrerons plus loin ...)
  - à chaque tour de boucle, l'instruction/séquence est d'abord exécutée, puis (en interne) l'incrémentation de indice est effectuée (`indice ← indice + 1`); on termine donc bien avec `indice > valFin`
  - la clause `par p` permet de spécifier le "pas d'incrémentation"; par défaut, il vaut 1, mais il peut prendre n'importe quelle valeur entière, y compris négative

Ces déclarations et surtout ces procédures sont loin d'être aussi 'élégantes' qu'elles le devraient :

- la variable indice est déclarée Entier alors que les seules valeurs autorisées (cfr. le contenu des procédures) est l'intervalle 1 à 11 ou 0 à 10 (selon les procédures)
- il y est fait référence à plusieurs reprises et explicitement aux valeurs (littérales) inférieure et supérieure des indices du tableau, ce qui les rend dépendantes du tableau (elles doivent être réécrites pour chaque tableau particulier et/ou modifiées lorsque les dimensions du tableau changent)
- on va donc exploiter une possibilité offertes sur les types de données scalaires énumérés (Entier, Caractère) : la possibilité de déclarer une variable sur un intervalle

On obtient quelque chose de déjà plus correct ci-dessous, par l'usage de constantes ...

```

Constante bInf = 1, bSup = 10           # bornes utiles
Variable tbl : Tableau[bSup] de Entier # variable-tableau
      indice : bInf-1 .. bSup+1       # indice de position de cellule

```

réécrivons en conséquence les différentes formes d'itérateurs d'initialisation :

```

Procédure initialiser1a :
Début
  indice ← bInf
  tantQue non (indice > bSup)
    t[indice] ← 0
    indice ← indice + 1
  finTQ
Fin

```

```

Procédure initialiser1b :
Début
  indice ← bInf - 1
  tantQue non (indice = bSup)
    indice ← indice + 1
    t[indice] ← 0
  finTQ
Fin

```

```

Procédure initialiser2a :
Début
  indice ← bInf
  Répéter
    t[indice] ← 0
    indice ← indice + 1
  Jusque indice > bSup
Fin

```

```

Procédure initialiser2b :
Début
  indice ← bInf - 1
  Répéter
    indice ← indice + 1
    t[indice] ← 0
  Jusque indice = bSup
Fin

```

idem avec l'itérateur « fermé » Pour

```

Procédure initialiser3 :
Début
  Pour indice de bInf à bSup Faire
    t[indice] ← 0
  finPr
Fin

```

Ces itérateurs basiques peuvent être utilisés pour l'initialisation d'un tableau comme ci-dessus, mais également pour son chargement (depuis le clavier, un fichier ...) ou pour son affichage (à l'écran ou sur listing), ou encore pour toute opération de modification de (tout) son contenu ... il suffit simplement de remplacer l'instruction d'initialisation `t[indice] ← 0` par l'instruction (ou la séquence d'instructions) nécessaire, p.ex. :

- *charger (une cellule) du tableau depuis le clavier :*

```

écrire("introduire l'élément d'indice ", indice)
lire(t[indice])

```
- *afficher le contenu (d'une cellule) du tableau :*

```

écrire("l'élément d'indice ", indice, " est ", t[indice])

```
- *modifier le contenu (d'une cellule) du tableau (tout multiplier par 1.06, p.ex. ... des prix tvaC)*

```

t[indice] ← t[indice] * 1.06

```
- *exploiter le contenu (d'une cellule) du tableau*

```

variable ← (t[indice] * sin(t[indice - 1]))

```

Autre exemple faisant appel à un itérateur basique : on effectue en laboratoire une série de 10 mesures de tensions (entre 0 et 220V) au moyen d'un voltmètre ; on désire connaître la plus grande mesure, la plus petite ainsi que la moyenne des mesures (version très procédurale)

```

Algorithme mesurerTensions :
Constante bInf = 1, bSup = 10                                # bornes
Variable t : Tableau[bSup] de Réel                            # tableau
    mesureMin, mesureMax, mesureMoy : Réel                    # résultats

Procédure ajouterMesure(ind : Entier) :                      # paramètre = indice courant
Variable mesure : Réel # variable locale pour la saisie/validation
Début
    Répéter
        écrire("mesure n° ", indice, " (0 à 220V) : ")
        lire(mesure)
    Jusqu'à mesure entre 0.0 et 220.0 # boucle de validation
    t[ind] ← mesure # valide, enregistrer dans le tableau
Fin

Procédure chargerTableau :
Variable indice : Entier # variable locale pour itérer le tableau
Début
    Pour indice de bInf à bSup Faire
        ajouterMesure(indice) # !! invocation avec passage de paramètre
    finPr
Fin

Procédure afficherTableau :
Variable indice : Entier # variable locale pour itérer le tableau
Début
    Pour indice de bInf à bSup Faire
        écrire("mesure n° ", indice, " est : ", t[indice])
    finPr
Fin

Procédure calculerRésultats :
Variable indice : Entier # variable locale pour itérer le tableau
    sommeMesures : Réel # variable locale pour calculer moyenne
Début
    sommeMesures ← 0.0
    mesureMin ← t[bInf] # initialisation du minimum
    mesureMax ← t[bInf] # initialisation du maximum
    Pour indice de bInf + 1 à bSup Faire # remarquer l'indice de départ !
        Si t[indice] < mesureMin Alors mesureMin ← t[indice] finSi
        Si t[indice] > mesureMax Alors mesureMax ← t[indice] finSi
        sommeMesures ← sommeMesures + t[indice]
    finPr
    mesureMoy ← sommeMesures / (bSup - bInf + 1) # nombre d'éléments
Fin

Procédure afficherRésultats :
Début
    afficherTableau
    écrire("nombre de mesures : ", bSup - bInf + 1)
    écrire("mesure minimale : ", mesureMin)
    écrire("mesure maximale : ", mesureMax)
    écrire("mesure moyenne : ", mesureMoy)
Fin

Début
    chargerTableau
    calculerRésultats
    afficherRésultats
Fin.

```

Remarque : les versions avec d'autres types de boucles sont laissées au lecteur ; l'écriture ci-dessus illustre des itérateurs (en utilisant beaucoup, à but pédagogique) ; mais il est possible de simplifier cet algorithme, et en particulier de déterminer les mesures min et max 'au vol', au fur et à mesure de la saisie (alternative laissée au lecteur comme exercice)



g) ITÉRATEURS POUR LA VALIDATION/VÉRIFICATION D'UN TABLEAU

On ne saurait pas dresser ici la liste complète des types de vérifications de contenu d'un tableau (cela ne se limite pas à la simple validité de chaque élément, cela dépendra de chaque cas d'espèce posé par le problème dont la solution nécessite l'usage d'un tableau) ...


Abordons la problématique d'une validation plus globale (sur l'ensemble des éléments du tableau les uns par rapport aux autres, donc), ce que l'on pourrait désigner par « vérification d'ordonnement de tableau ».

Pour rappel : une relation d'ordre est basée sur la comparaison, c.-à-d. sur les opérateurs  $=$ ,  $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $\neq$  (dont l'utilisation renvoie on s'en souviendra une valeur de type Logique : vrai ou faux).

Le tri (croissant ou décroissant) n'est donc qu'un ordonnancement particulier, basé sur les seuls opérateurs  $>$  (tri croissant) ou  $<$  (tri décroissant) ; nous élargirons ici cette notion simpl(ist)e, mais d'un très large usage, à tout type d'ordre (p.ex., un problème pourrait exiger que les éléments d'un tableau respectent la contrainte : *à part les deux premiers, tout élément doit être la somme de tous ceux qui le précèdent*)

Exemple simple : vérification d'un simple ordre croissant entre les éléments

1	3	4	6	5	8	10	11	5	13
---	---	---	---	---	---	----	----	---	----



Le principe d'une telle vérification est le suivant : il suffit – en le parcourant - de trouver (au moins) un couple d'éléments consécutifs du tableau ne respectant pas la relation d'ordre pour conclure au non respect de cette relation sur tableau ; si on atteint la borne supérieure du tableau sans avoir rencontré un tel couple, on peut conclure au respect de la relation pour l'ensemble du tableau.

Algorithme `verifierOrdreBasique` :

Constante `bInf` = 1, `bSup` = 10

Variable `t` : Tableau[`bSup`] de Entier

`estOrdonné` : Logique

Procédure `chargerTableau` :

Variable `indice` : `bInf` .. `bSup`

Début

**Pour** `indice` de `bInf` à `bSup` **Faire**

`lire(t[indice])`

**finPr**

Fin

Procédure `vérifierOrdreBasique` :

Variable `indice` : `bInf` .. `bSup`

Début

`estOrdonné`  $\leftarrow$  vrai

# soyons optimistes

**Pour** `indice` de `bInf` à `bSup - 1` **Faire** # s'arrêter à temps ! <sup>66</sup>

Si `t[indice] > t[indice + 1]` Alors `estOrdonné`  $\leftarrow$  faux finSi

**finPr**

Fin

Début

`chargerTableau`

`vérifierOrdreBasique`

Si `estOrdonné` Alors écrire("ordonné") Sinon écrire("non ordonné") finSi

Fin

L'inconvénient de cette écriture, c'est qu'elle utilise l'itérateur "fermé" qui oblige de parcourir systématiquement tout le tableau depuis `bInf` jusque `bSup` ; il est un peu ridicule (et cher en performance) de continuer systématiquement l'exploration du tableau dès lors que l'on a rencontré un couple d'éléments consécutifs ne respectant pas la relation d'ordre (il suffit d'imaginer un tableau de un million d'éléments dont les deux premiers ne sont pas ordonnés !).

<sup>66</sup> alternative : Pour `indice` de `bInf + 1` à `bSup` Si `t[indice] < t[indice - 1]` Alors ...

Améliorons l'algorithme en arrêtant l'itérateur dès que la relation d'ordre n'est pas respectée ; pour cela, il faut abandonner l'itérateur "fermé" et se rabattre sur les 'bonnes vieilles' boucles : elles permettent en effet d'ajouter la variable logique dans la condition d'arrêt (avec le Tableau, on rencontre un ensemble de problèmes présentant deux conditions d'arrêt : ici on utilisera une boucle 'répéter' ; le lecteur est prié de refaire le même travail mais avec une boucle 'tantQue' et donc une condition 'de continuation') :

```

Constante bInf = 1, bSup = 10
Variable t : Tableau[bSup] de Entier
        estOrdonné : Logique

Procédure chargerTableau :
Début
    ... # remplir le tableau, peu importe comment
Fin

Procédure vérifierTableauAvancé :
Variable indice : bInf .. bSup
Début
    estOrdonné ← vrai # soyons optimistes
    indice ← bInf
    Répéter
        Si t[indice] > t[indice + 1] Alors estOrdonné ← faux finSi
        indice ← indice + 1
    Jusque (indice > bSup ou non estOrdonné) # itérateur avec interruption
Fin

Début
    chargerTableau
    vérifierTableauAvancé
    Si estOrdonné Alors écrire('ordonné') Sinon écrire('non ordonné') finSi
Fin.
```

Faisons un bref retour sur l'itérateur "fermé" (boucle 'Pour' ...); a priori, il n'existe que la version chère du parcours complet ...

```

Procédure verifierTableauPour1 :
Début
    estOrdonné ← vrai # soyons optimistes
    Pour indice de bInf à bSup - 1 Faire # attention : bSup - 1 !!
        Si t[indice] > t[indice + 1] Alors estOrdonné ← faux finSi
    finPr # on a tout parcouru !
Fin
```

... mais à y regarder de plus près, comme l'indice n'atteindra pas la borne supérieure bSup (on a en effet écrit Pour indice de bInf à **bSup - 1**), on peut donc utiliser cette particularité pour forcer l'indice à la valeur bSup lorsqu'on rencontre un couple d'éléments consécutifs non ordonnés.

La boucle, ayant à ce moment un indice dont la valeur (bSup) est supérieure à sa valeur maximale (bSup - 1), est forcée de s'arrêter !

A remarquer également que la variable logique n'est dès lors plus nécessaire et doit être remplacée par un test sur l'indice (en sortie de boucle) :

```

Procédure verifierTableauPour2 :
Début
    Pour indice de bInf à bSup - 1 Faire # attention : bSup - 1 !!
        Si t[indice] > t[indice + 1] Alors indice ← bSup finSi # forçage
    finPr
Fin
...
Début
    chargerTableau
    verifierTableauPour2
    Si indice = bSup
        Alors écrire("non ordonné") Sinon écrire("ordonné") finSi
Fin.
```

Toute méthode présente des inconvénients : on le verra plus loin, il est souvent intéressant de connaître l'indice du tableau sur lequel l'itérateur s'est arrêté ; la méthode ci-dessus détruit cet indice (en le forçant à bSup pour pouvoir s'arrêter) !

Aussi certains concepteurs de langages proposant la boucle 'Pour' (sous une forme ou une autre...) se sentent-ils obligés de doter celle-ci d'une instruction de 'sortie forcée' (du genre stopperPour, quitterPour, stopperBoucle, quitterBoucle, ... équivalent ☹ d'un « GoTo à l'instruction suivant la fin de la boucle »)

Exemple ci-dessous (bien entendu, la variable logique doit faire sa réapparition ...)

```
Procédure verifierTableauPour3 :
Début
  estOrdonné ← vrai                                # soyons optimistes
  Pour indice de bInf à bSup - 1 Faire              # attention : bSup - 1 !!
    Si t[indice] > t[indice + 1] Alors
      estOrdonné ← faux
      quitterPour # sortie forcée, un vrai crime contre l'algorithmique ! ☹
    finSi
  finPr
Fin
```

☹ voilà, on l'a montré; maintenant on l'oublie à tout jamais ! ☹ <sup>67</sup>

#### h) TYPE TABLEAU ABSTRAIT : ADT & API

Pour pouvoir rédiger les outils procéduraux (API) permettant de considérer qu'une structure de Tableau est une valeur sur laquelle on pourra effectuer des opérations globalement, il faut lui conférer un statut de type abstrait,

par exemple pour le problème de gestion de cotes, on crée un nouveau type abstrait (un ADT) comme suit :

```
Constante nbCotes = 50
Type TtbCotes = Tableau[nbCotes] de Réel
```

de la sorte, on peut créer facilement plusieurs variables de ce type (comme on a pu le faire avec les types scalaires prédéfinis et les ADT Enregistrement) :

```
Variable tbCotes1TL, tbCotes1TM : TtbCotes
```

Une fois de plus, le gain est significatif : désormais, puisqu'elles se réclament d'un type existant, des variables telles que tbCotes1TL et tbCotes1TM possèdent une 'vraie' valeur

- on récupère d'un seul coup l'affectation globale et on pourra donc écrire :  

```
variable tbCotes : TtbCotes
...
tbCotes1TL ← tbCotes    # parce qu'elles possèdent le même type !
```
- les opérateurs de 'haut niveau' API pourront être rédigés à l'aide des procédures et fonctions paramétrées (les paramètres et les valeurs de retour de fonctions doivent toujours être typés !); par exemple ci-dessous, nous allons écrire une nouvelle 'affectation externe' pour des variables de type TtbCotes (on part de l'idée que si on peut écrire tbCotes1TL ← tbCotes, alors tbCotes peut être remplacée par une fonction ....

<sup>67</sup> la simple honnêteté intellectuelle nous oblige à mentionner que dans un langage de programmation tel que le C, l'usage du forçage de boucle (au moyen de l'instruction break) est considéré comme un bon style de programmation ! ce n'est pas le cas ni en Pascal ni en algorithmique pseudo-code où il est considéré comme non structuré

```

Algorithme Cotations :
Constante coteMin = 0.0, coteMax = 20.0, nbCotes = 50
Type TtbCotes = Tableau[nbCotes] de Réel
Variable tbCotes1TL, tbCotes1TM : TtbCotes           # variables globales de l'algo

Fonction coteValide(c : Réel) : Logique :
Début
    renvoie(c entre coteMin et coteMax)           # comparaison : résultat = vrai/faux
Fin

Fonction lireCote() : Réel :                       # fournit une cote validée
Variable c : Réel
Début
    Répéter
        lire(c)                                   # invite non illustrée
    Jusqu'à coteValide(c)                         # mais avec validation
Fin

Fonction lireCotes() : TtbCotes :                 # API : une 'valeur' tableau de cotes
Variable tb : TtbCotes                           # variable locale de type tableau de cotes
    indice : Entier                               # variable locale itérateur
Début
    Pour indice de 1 à nbCotes Faire
        tb[indice] ← lireCote()                   # encoder chaque cote à son tour ....
    finPr
    renvoie(tb)                                   # fournit la valeur au demandeur
Fin

Début # séquence principale
...
    tbCotes1TL ← lireCotes()                     # affectation interne globale généralisée
...
Fin

```

Une fois de plus, on est confronté à un dilemme : lors de la rédaction d'une API, comment choisir entre les fonctions paramétrées et les procédures paramétrées ? ...

Dès lors qu'il s'agit de traiter une seule valeur-structure (comme ici un tableau de cotes), on peut opter pour

- une généralisation de l'affectation interne au moyen d'une fonction (qui figurera à droite du symbole d'affectation)

```

Algorithme Cotations :
Constante coteMin = 0.0, coteMax = 20.0, nbCotes = 50
Type TtbCotes = Tableau[nbCotes] de Réel
Variable tbCotes1TL, tbCotes1TM : TtbCotes  # variables globales de l'algo

Fonction lireCotes() : TtbCotes :          # renvoie un tableau de cotes
Variable tb : TtbCotes                      # variable locale tableau de cotes
    indice : Entier                          # variable locale itérateur
Début
    Pour indice de 1 à nbCotes Faire          # ni invite, ni validation
        lire(tb[indice])                     # encoder chaque cote à son tour...
    finPr
    renvoie(tb)                             # fournit la valeur au demandeur
Fin

Début
...
    tbCotes1TL ← lireCotes()  # affectation interne globale généralisée
    tbCotes1TM ← lireCotes()
...
Fin

```

- une généralisation de l'affectation externe (de la procédure lire()) au moyen d'une procédure paramétrée, MAIS la structure de tableau 'à lire' doit être passée 'par référence' (puisque son état sera modifié)

```

Algorithme Cotations :
Constante coteMin = 0.0, coteMax = 20.0, nbCotes = 50
Type TtbCotes = Tableau[nbCotes] de Réel
Variable tbCotes1TL, tbCotes1TM : TtbCotes  # variables globales de l'algo

Procédure lireCotes(ref tb : TtbCotes) :    # modifie un tableau de cotes
Variable indice : Entier                      # variable locale itérateur
Début
    Pour indice de 1 à nbCotes Faire          # ni invite, ni validation
        lire(tb[indice])                     # encoder chaque cote à son tour...
    finPr
Fin

Début
...
    lireCotes(tbCotes1TL)                    # affectation externe globale généralisée
    lireCotes(tbCotes1TM)
...
Fin

```

### i) TAILLE PHYSIQUE ET TAILLE LOGIQUE D'UN TABLEAU PARTIEL

Il serait faux de croire que les tableaux déclarés dans les programmes informatiques sont toujours utilisés « à pleine charge » (c.-à-d. que toutes leurs cases sont occupées).

Le plus souvent il n'en est rien ; on fait une déclaration de tableau dont la taille est 'confortable' vis-à-vis du problème posé ; en d'autres termes, on déclare souvent 'plus grand', de manière à avoir une marge de sécurité ... si on prévoit d'enregistrer environ une centaine de mesures, on déclarera un tableau de – disons – 120 éléments (de là à déclarer systématiquement le 'plus grand' tableau possible – toute la mémoire disponible - il y a de la marge !)

Il y aura donc des cases inutilisées et la question est évidemment de savoir faire la différence entre les cases 'utiles' et les cases 'inutiles' ...

[illegible]

Une solution simple et élégante est utilisée depuis toujours :

- une constante définit le nombre maximum d'éléments du tableau : nous la qualifierons de taille physique et elle est utilisée lors de la déclaration pour réserver l'espace du tableau en mémoire et comme valeur d'indice à ne jamais dépasser lors du chargement
- on remplit et on exploite la partie inférieure (d'indices faibles, à partir de 1) du tableau (le tableau devient ainsi – du point de vue algorithmique - un tableau partiel)
- et on conserve et on gère soigneusement – par une variable supplémentaire – la plus haute valeur de l'indice atteinte lors du chargement; nous la qualifierons de taille logique

[illegible]

La taille physique correspondra donc à la constante PSup, et la taille logique à la variable LSup, comme suit :

```

Constante PSup = 20 # définition de la taille physique du tableau
Type TtbMesures = Tableau[PSup] de Entier # type Tableau de mesures
Variable tbM : TtbMesures # variable tableau de mesures
         LSup : Entier # variable indiquant la taille logique

```

Comment charger un tableau partiel ? Deux cas sont possibles :

- 1) on connaît le nombre d'éléments à l'avance (p.ex. avant d'effectuer les mesures, décider de leur nombre); on peut donc affecter directement la variable LSup et écrire une boucle de chargement 'fermée' ayant LSup comme borne supérieure :

```

Procédure chargerTableau(ref t : TtbMesures, ref b : Entier) :
Variable indice : Entier      # indice d'itérateur
Début
  lire(b)                     # lire le nombre d'éléments à charger
  Si b <= PSup Alors          # prudence quand même !!
    Pour indice de 1 à b Faire # boucle fermée ...
      lire(t[indice])          # lecture simple et affectation directe
    finPr
  Sinon
    # traitement d'erreur
  finSi
Fin

Début # séquence principale
...
  chargerTableau(tbM, LSup)    # chargement par procédure paramétrée
...
Fin

```

- 2) le nombre d'éléments n'est déterminé qu'au fur et à mesure du chargement (on encode au fur et à mesure qu'on effectue les mesures ... *tant qu'il y en a* ...)

Plus question d'itérateur fermé, cette fois; il faut revenir aux boucles traditionnelles, avec une préférence pour un tantQue, seul à même de ne pas charger le tableau du tout ! (on imagine ici un encodage avec une invite du genre "*mesure (-9999 = stop)* : ")

```

Procédure chargerTableau(ref t : TtbMesures, ref b : Entier) :
Variable m : Réel          # variable locale pour l'encodage
      fini : Logique       # état de l'encodage
Début
  b ← 0                    # au départ le tableau est vide !
  écrire("mesure (-9999 = stop) : ")
  lire(m)
  fini ← (m = -9999)       # arrêt utilisateur
  tantQue non fini Faire   # si on continue ...
    b ← b + 1              # définir la nouvelle borne supérieure logique
    t[b] ← m               # ... et y placer la valeur lue
    fini ← (b = PSup)      # arrêt 'interne' (tableau physique rempli)
  Si non fini Alors
    écrire("mesure (-9999 = stop) : ")
    lire(m)                # mesure suivante
    fini ← (m = -9999)     # arrêt utilisateur
  finSi
finTQ
Fin

Début # séquence principale
...
chargerTableau(tbM, LSup)  # chargement par procédure paramétrée
...
Fin

```

Conseil rédactionnel : il est à conseiller de passer par une variable locale d'encodage (et de ne pas encoder directement dans le tableau); en effet , l'écriture suivante est particulièrement lourde et dangereuse (on oublie trop facilement quelque chose ...)

```

Procédure chargerTableau(ref t : TtbMesures, ref b : Entier) :
Variable fini : Logique    # état de l'encodage
Début
  b ← 1                    # il faut pouvoir accéder à la première cellule !
  écrire("mesure (-9999 = stop) : ")
  lire(t[b])               # lire t[1] donc ...
  fini ← (t[b] = -9999)    # arrêt utilisateur
  tantQue non fini Faire   # si on continue ...
    b ← b + 1              # définir la nouvelle borne supérieure logique
    fini ← (b > PSup)      # arrêt 'interne' (tableau physique rempli)
  Si non fini Alors
    écrire("mesure (-9999 = stop) : ")
    lire(t[b])             # mesure suivante
    fini ← (t[b] = -9999)  # arrêt utilisateur
  finSi
finTQ
  b ← b - 1                # ne pas oublier de décrémenter !!!
Fin

```

Bien entendu, on peut être moins prudent et considérer que l'utilisateur encodera au moins une mesure, une logique d'encodage basée sur un Répéter ... Jusque ... est dès lors possible; mais elle est laissée aux boins soins du lecteur à tritre d'exercice ...

Pourquoi avoir eu recours à des procédures et non à des fonctions ? parce que cette fois deux valeurs sont modifiées : la variable-tableau et la variable-taille (seule une procédure peut modifier plusieurs variables qui lui sont transmises alors par référence) ... patience ...

j) ITÉRATEURS (BOUCLES) IMBRIQUÉS

Dans pas mal d'algorithmes, on est amenés à parcourir simultanément un tableau avec deux itérateurs distincts ; cela ne pose pas plus de problème que d'imbriquer deux boucles dans les algorithmes généraux, à ceci près qu'il faut correctement définir les bornes supérieure et inférieure de chaque itérateur (habituellement celles du plus 'intérieur' dépendent fortement de celles du plus 'extérieur')

Exemple : vérifier un ordonnancement particulier d'un tableau : chaque élément (à partir du deuxième) doit être strictement supérieur à la somme de tous ceux qui le précèdent :

3	6	10	21	45	89	97	125	325											
---	---	----	----	----	----	----	-----	-----	--	--	--	--	--	--	--	--	--	--	--

Algorithme vérifierOrdreTableau

Constante PSup = 20 # définition de la taille physique du tableau

Type **TtbNombres** = Tableau[PSup] de Entier # type Tableau de entiers

Variable **tbN** : TtbNombres # variable tableau de nombres

**LSup** : Entier # variable indiquant la taille logique

ordonné : Logique # variable globale : état du tableau

Procédure chargerTableau(ref t : TtbNombres, ref b : Entier) :

Début

...

Fin # ici le tableau est (partiellement) chargé et LSup donne sa taille réelle

Procédure validerTableau(**t** : TtbNombres, **b** : Entier) : # par ref !!

Variable ind1, ind2 : Entier # variables locales pour itération

somme : Entier # variable locale pour vérification

Début

ordonné ← vrai # soyons optimistes !

ind1 ← 2

**tantQue** estOrdonné et non (ind1 > **b**) # itérateur "extérieur"

somme ← 0.0 # initialiser

**Pour** ind2 de 1 à ind1 - 1 **Faire** # itérateur "intérieur", ok pour un Pour

somme ← somme + t[ind2] # totaliser

**finPr** # somme des éléments avant élément courant

ordonné ← (t[ind1] > somme) # réévaluer condition d'arrêt

ind1 ← ind1 + 1 # élément courant = suivant !

**finTQ**

Fin

Début

chargerTableau(tbN, Lsup) # transmis par référence : modification

validerTableau(tbN, Lsup) # transmis par valeur : pas touche au contenu !

Si ordonné Alors écrire('ok') Sinon écrire('ko') finSi

Fin

Remarques : une fois de plus,

- l'écriture des versions alternatives (et plus élégantes peut-être du point de vue algorithmique formel) avec d'autres types de boucles est laissée aux bons soins du lecteur ...
- la version ci-dessus utilise deux itérateurs pour les besoins de la description de l'imbrication de boucles, mais il est possible d'écrire un algorithme équivalent avec un seul itérateur ... à vous de jouer ...



k) TABLEAUX COUPLÉS EXPLOITÉS "EN PARALLÈLE"

Pour introduire la suite, examinons une variation sur le thème de la *gestion de cotes* évoqué précédemment : on ajoute au besoin de mémoriser les cotes dans un tableau la nécessité d'associer à chacune le nom de l'étudiant; on doit donc déclarer deux (types) de tableaux :

```

Algorithme Cotations :
Constante coteMin = 0.0, coteMax = 20.0, nbCotes = 50
Type TtbCotes = Tableau[nbCotes] de Réel    # pour les cotes
Type TtbNoms = Tableau[nbCotes] de Texte   # pour les noms
Variable tbCotes1TL, tbCotes1TM : TtbCotes
        tbNoms1TL, tbNoms1TM : TtbNoms      # variables globales de l'algo

Fonction coteValide(c : Réel) : Logique :
Début
    renvoie(c entre coteMin et coteMax)      # comparaison : résultat = vrai/faux
Fin

Procédure lireCote(ref tC : TtbCotes, ref tN : TtbNoms, i : Entier) :
# lit et mémorise le ième couple nom + cote
Variable c : Réel, n : Texte                # cote et nom pour la saisie
Début
    lire(n)                                  # lecture nom (invite non illustrée)
    Répéter
        lire(c)                              # lecture cote (invite non illustrée)
    Jusqu'à coteValide(c)                    # mais avec validation
    tN[i] ← n                                # mémoriser nom
    tC[i] ← c                                # mémoriser cote
Fin

Procédure lireCotes(ref tC : TtbCotes, ref tN : TtbNoms) : # par référence !
# lit et mémorise les nbCotes cotes et noms
Variable indice : Entier                    # variable locale itérateur
Début
    Pour indice de 1 à nbCotes Faire
        lireCote(tC, tN, indice)             # lire et mémoriser tous les noms et cotes
    finPr
Fin

Procédure afficherCotes(tC : TtbCotes, tN : TtbNoms) :      # par valeur !
Variable indice : Entier                    # variable locale itérateur
Début
    Pour indice de 1 à nbCotes Faire
        écrire(tN[indice], " ", tC[indice], "/20")
    finPr
Fin

Début # séquence principale
...
    lireCotes(tbCotes1TL, tbNoms1TL) # affectation externe globale généralisée
    lireCotes(tbCotes1TM, tbNoms1TM)

    afficherCotes(tbCotes1TL, tbNoms1TL)
    afficherCotes(tbCotes1TM, tbNoms1TM)
...
Fin

```

Cette obligation de travailler dans deux tableaux distincts mais en parallèle (il y a une correspondance entre les éléments de même indice (un nom, une cote) alourdit considérablement le traitement : on perd la possibilité d'utiliser des fonctions et doit utiliser des procédures multiparamétrées ...et encore, on a simplifié le problème en se limitant à deux tableaux et en évitant d'utiliser des tableaux partiels (la gestion de la taille logique ajoute des contraintes supplémentaires ...)... c'est d'ailleurs un excellent exercice laissé aux bons soins du lecteur

Une fois de plus, il va falloir regarder cela d'un peu plus haut !

## I) CONCRÈTEMENT ...

### Fortran

Les tableaux font partie du langage dès ses origines; la déclaration s'effectue simplement en suffixant le nom générique du tableau par son nombre d'éléments au moyen de parenthèses (où à l'aide d'une clause spécifique DIMENSION)

```
REAL TB1(100)
DIMENSION INTEGER TB2(100)
INTEGER INDICE
```

Il est également possible d'utiliser une 'constante' tant pour la taille du tableau que pour contrôler ensuite ses itérateurs

```
INTEGER MAXELEM
DATA MAXELEM /100/
REAL TB1(MAXELEM)
INTEGER INDICE
```

Le langage comporte un itérateur spécifique (correspondant à la boucle Pour) ainsi qu'un opérateur d'accès (les parenthèses et non le couple classique []) : ainsi pour initialiser à 0 les cases des tableaux TB1 et TB2 (Pour indice de 1 à maxelem Faire tb1[indice] = 0.0; tb2[indice] = 0.0 finPr) on écrira

```
DO 10 INDICE = 1, MAXELEM
    TB1(INDICE) = 0.0
10  TB2(INDICE) = 0.0
```

l'instruction DO (faire) spécifie le numéro de la ligne du programme qui est la dernière du corps de la boucle

### Cobol

Présents également dès sa première version, les tableaux en Cobol sont très riches en possibilités de toutes sortes (il serait trop long de les évoquer ici). Basiquement la déclaration d'un tableau s'effectue à l'aide d'une clause spécifiant le type (PIC) et le nombre (OCCURS) d'éléments :

```
77 MAXELEM PIC 99 VALUE 100.
01 TB1.
   02 TB1EL PIC 9(4) OCCURS MAXELEM TIMES.
01 TB1.
   02 TB1EL PIC 9(4) OCCURS MAXELEM TIMES.
77 INDICE PIC 99.
```

L'itérateur est une boucle utilisant une procédure (paragraphe) contenant le corps de la boucle qui est commandée par une instruction (PERFORM VARYING) spécifiant l'indice de contrôle, sa valeur de départ, son pas (incrément) et c'est une condition tantQue (qui s'écrit UNTIL en Cobol !) qui se charge de la condition d'arrêt. L'accesseur aux éléments du tableau utilise les parenthèses, comme en Fortran

```
PERFORM BOUCLE VARYING INDICE FROM 1 BY 1
    UNTIL INDICE > MAXELEM.

BOUCLE.
    MOVE 0.0 TO TB1EL(INDICE).
    MOVE 0.0 TO TB2EL(INDICE).
```

Parmi les possibilités intéressantes figure la possibilité d'initialisation 'en une ligne' via une affectation globale sur le nom générique de la structure; tout ce qui précède peut donc être remplacé par

```
MOVE ALL 0.0 TO TB1, TB2.
```

## Pascal

Très proche du pseudo-code évidemment, Pascal permet aussi bien de déclarer des tableaux non typés que typés (seuls ceux-ci pourront être utilisés comme paramètres pour les fonctions et procédures). Le principe de leur déclaration (typée, ADT) est la suivante :

```
const nbCotes = 50;
type TtbCotes = array[1 .. nbCotes] of real;
```

Il suffit ensuite de déclarer les variables relevant de ce nouveau type

```
var tbC1, tbC2 : TtbCotes;
```

Pour l'initialisation, on rédigera une procédure paramétrée (passage par adresse) comportant un itérateur implémentant la boucle Pour (for); l'accesseur aux cellules est cette fois []

```
procedure pInitTableau(var t : TtbCotes);
var indice : integer;
begin
  for indice := 1 to nbCotes do
    t[indice] := 0.0
  end;
```

mais on peut préférer une fonction (qui renvoie un tableau initialisé)

```
function fInitTableau() : TtbCotes;
var indice : integer;
    t : TtbCotes;
begin
  for indice := 1 to nbCotes do
    t[indice] := 0.0;
  fInitTableau() ← t
end;
```

## C

Le C comporte la possibilité d'initialiser un tableau lors de sa déclaration (économie de code !); pour le reste, on trouve en gros les mêmes possibilités qu'en Pascal (sauf que la première cellule d'un tableau a nécessairement l'indice 0) : déclaration (abstraite)

```
#define NB_COTES 50
typedef float TtbCotes[50]; // type tableau de 50 réels

TtbCotes tb1 = {0.0}, tb2; // tb1 est initialisé entièrement, pas tb2
```

l'itérateur est une boucle for (petite usine du C) que l'on peut utiliser dans 'vraie' fonction d'initialisation

```
TtbCotes fInitTableau(void) {
  TtbCotes t;
  short indice;
  for (indice = 0; indice < NB_COTES; indice++)
    t[indice] = 0.0;
  return t;
}
```

ou dans une 'vraie' procédure : il faut savoir que les tableaux sont toujours passés par adresse, ce qui soulage la programmation (pas besoin des opérateurs d'adressage et d'indirection)

```
void pInitTableau(TtbCotes t) {
  short indice;
  for (indice = 0; indice < NB_COTES; indice++)
    t[indice] = 0.0;
}
```

m) EXERCICES

1. Lors de l'illustration de la vérification d'un simple ordre croissant entre les éléments d'un tableau

1	3	4	6	5	8	10	11	5	13
---	---	---	---	---	---	----	----	---	----

nous avons écrit la procédure suivante :

```

Procédure vérifierOrdreBasique :
Variable indice : Entier
Début
    estOrdonné ← vrai                                # soyons optimistes
    Pour indice de bInf à bSup - 1 Faire              # s'arrêter à temps ! 68
        Si t[indice] > t[indice + 1] Alors estOrdonné ← faux finSi
    finPr
Fin

```

or nous avons toujours 'milité' pour une affectation directe dans les variables logiques; que penser alors de cette écriture alternative 'puriste' ?

```

Procédure vérifierOrdreBasique :
Variable indice : Entier
Début
    estOrdonné ← vrai                                # soyons optimistes
    Pour indice de bInf à bSup - 1 Faire              # s'arrêter à temps !
        estOrdonné ← (t[indice] > t[indice + 1])
    finPr
Fin

```

2. Rédigez un algorithme construisant un nouveau tableau, à partir de deux tableaux de même taille (nombre d'éléments) préalablement chargés d'entiers.

Chaque élément du nouveau tableau sera la somme des éléments de même indice des deux tableaux de départ.

Exemple

Tableau 1 :

4	8	7	9	1	5	4	6
---	---	---	---	---	---	---	---

Tableau 2 :

7	6	5	2	1	3	7	4
---	---	---	---	---	---	---	---

Tableau à constituer :

11	14	12	11	2	8	11	10
----	----	----	----	---	---	----	----

Faites une version ADT/API utilisant une fonction et une autre utilisant une procédure

3. Que fait cet algorithme ?

```

Variable t : Tableau [10] de Entier
    indice : Entier
Début
    Pour indice de 1 à 10 Faire
        t[indice] ← indice * indice
    finPr
    Pour indice de 1 à 10 Faire
        écrire(t[indice])
    finPr
Fin

```

<sup>68</sup> alternative : Pour indice de bInf + 1 à bSup Si t[indice] < t[indice - 1] Alors ...

#### 4. Que fait cet algorithme ?

```

Constante bas=1, haut = 8
Variable suite : Tableau[8] de Entier
        indice : Entier
Début
    suite[bas] ← 1
    suite[bas + 1] ← 1
    Pour indice de bas + 2 à haut Faire
        suite[indice] ← suite[indice - 1] + suite[indice - 2]
    finPr
    Pour indice de bas à haut
        écrire(t[indice])
    finPr
Fin

```

#### 5. Comment utiliser un tableau pour stocker un polynôme (en vue d'une évaluation ultérieure) ? le choix judicieux des indices est la clé de cette implémentation.

*Exemple*

Le polynôme  $5.x^3 + 9.x^2 + 12.x - 7$  peut être stocké de la sorte :

-7	12	9	5				
----	----	---	---	--	--	--	--

Rédiger un algorithme pour calculer sa valeur en  $x = 5$ , puis en  $x = -10$

Faites une version ADT/API utilisant une seule fonction : *Poly(...)* : Réel

#### 6. Rédiger un algorithme qui inverse l'ordre des éléments d'un tableau dont on suppose qu'il a été préalablement rempli.

*Exemple*

Tableau au départ :

41	83	17	99	61	52	43	66
----	----	----	----	----	----	----	----

Tableau à l'arrivée :

66	43	52	61	99	17	83	41
----	----	----	----	----	----	----	----

Faites une version ADT/API utilisant une procédure : *invTabl(...)*

#### 7. Rédiger un algorithme permettant à l'utilisateur de saisir un nombre quelconque de valeurs réelles comprises entre $-10^3$ et $+10^3$ , qui devront être conservées dans un tableau.

L'utilisateur devra spécifier (via le clavier) le nombre de valeurs qu'il compte saisir.

Il effectuera ensuite cette saisie.

Enfin, une fois la saisie terminée, on affichera le nombre de valeurs négatives et le nombre de valeurs positives.

On considère que la mémoire ne permet pas de stocker plus de 1000 valeurs réelles.

Effectuez à tous les niveaux toutes les validations nécessaires

Rédigez une version ADT/API complète : fonctions et procédures éventuelle et séquence principale les invoquant



11. Rédiger un algorithme qui permet de déterminer si une chaîne de caractères entrée par l'utilisateur au clavier est un palindrome ou non.  
On appelle palindrome un mot qui est identique une fois écrit à l'envers (p.ex. 'radar').  
Utilisez un tableau de caractères (une lettre du mot par case du tableau).
-





## 5.4. STRUCTURES MIXTES

### a) INTRODUCTION

Certains problèmes – comme celui terminant le chapitre précédent - devraient être traités à l'aide de plusieurs tableaux exploités en parallèle; ce n'est pas la manière ni la plus simple, ni la plus élégante de procéder ...

Commençons par nous interroger sur ce qu'est une cotation : n'y a-t-il pas ici (au minimum) deux facettes complémentaires : un nom et une cote, et n'ont-elles pas une nature et un type différent (un Texte, un Réel) ?

Il est temps de se rappeler que la structure de données Enregistrement existe, et de voir ce qu'elle peut apporter.

Si l'on définit un ADT comme suit

```
Type TEtuCote = Enregistrement
    nom : Texte
    cote : Réel
    finEnr
```

on dispose d'une association bien plus claire et solide entre les composants `nom` et `cote`, et surtout, toute donnée de type `TEtuCote` possède une (super)valeur ! on peut donc réintégrer les fonctions comme outil d'API, ce qui est plus léger

```
Fonction lireCote() : TEtuCote :
Variable nc : TEtuCote
    n : Texte, c : Réel          # nom et cote pour la saisie
Début
    lire(n)                     # lecture nom (invite non illustrée)
    Répéter
        lire(c)                 # lecture cote (invite non illustrée)
    Jusqu'à coteValide(c)       # mais avec validation
    nc.nom ← n                  # construire enregistrement, début ...
    nc.cote ← c                 # ... fin
    renvoie(nc)                # renvoi enregistrement au demandeur
Fin
```

on va à présent remplacer les deux tableaux par un seul : un tableau dont les éléments (les valeurs !) sont des enregistrements de type `TEtuCote`

on pourrait commencer par ceci ...

```
Constante coteMin = 0.0, coteMax = 20.0, nbCotes = 50
Type TEtuCote = Enregistrement
    nom : Texte
    cote : Réel
    finEnr
Type TtbEtuCotes = Tableau[nbCotes] de TEtuCote
Variable tbCotation1TL, tbCotation1TM : TtbEtuCotes
```

... et faire dès à présent quelques exercices de nommage (on en aura besoin) en utilisant conjointement les deux accesseurs `[ ]` (côté Tableau) et `.` (côté Enregistrement)

Il s'agit ici d'un tableau d'enregistrements : les accesseurs s'utilisent de l'extérieur (ici le nom générique du tableau d'abord) vers l'intérieur (ici les champs de l'enregistrement ensuite)

donc pour la variable `tbCotation1TL`, la première (valeur de) cotation a pour nom `tbCotation1TL[1]` (première cellule du tableau), l'étudiant concerné a pour nom `tbCotation1TL[1].nom` et sa cote est `tbCotation1TL[1].cote` (champs constitutifs de cette première cellule)

b) RENFORCEMENT DE LA NOTION DE VALEUR

Écrivons une première version de cette API en privilégiant les fonctions (donc les valeurs !)

```

Constante coteMin = 0.0, coteMax = 20.0, nbCotes = 50
Type TÉtuCote = Enregistrement
    nom : Texte
    cote : Réel
    finEnr
Type TtbÉtuCotes = Tableau[nbCotes] de TÉtuCote
Variable tbCotation1TL, tbCotation1TM : TtbÉtuCotes

Fonction coteValide(c : Réel) : Logique :
Début
    renvoie(c entre coteMin et coteMax)    # comparaison : résultat = vrai/faux
Fin

Fonction lireÉtuCote() : TÉtuCote : # fournit une valeur (1 enregistrement) !!
Variable nc : TÉtuCote
    n : Texte, c : Réel # nom et cote pour la saisie
Début
    lire(n)                                # lecture nom (invite non illustrée)
    Répéter
        lire(c)                            # lecture cote (invite non illustrée)
    Jusqu'à coteValide(c)                  # mais avec validation
    nc.nom ← n                            # construire enregistrement, début ...
    nc.cote ← c                            # ... fin
    renvoie(nc)                           # renvoi enregistrement au demandeur
Fin

Fonction lireÉtuCotes() : TtbÉtuCotes : # fournit une valeur (1 tableau)
# lit et mémorise les nbCotes cotes et noms
Variable tec : TtbÉtuCotes                # tableau local à remplir
    indice : Entier                        # variable locale itérateur
Début
    Pour indice de 1 à nbCotes Faire
        tec[indice] ← lireÉtuCote()      # lire et mémoriser tous les noms et cotes
    finPr
    renvoie(tec)                           # renvoi du tableau au demandeur
Fin

Procédure afficherÉtuCotes(tec : TtbÉtuCotes) : # un seul paramètre
Variable indice : Entier
Début
    Pour indice de 1 à nbCotes Faire
        écrire(tec[indice].nom, tec[indice].cote)
    finPr
Fin

Début # séquence principale
...
    tbCotation1TL ← lireÉtuCotes()      # affectation interne globale généralisée
    tbCotation1TM ← lireÉtuCotes()
    afficherÉtuCotes(tbCotation1TL)      # un seul paramètre, passé par valeur
    afficherÉtuCotes(tbCotation1TM)
...
Fin

```

pas mal ! simplicité et élégance ... mais ... nous avons supposé que les deux tableaux de cotations étaient remplis 'physiquement' (nbCotes chacun) ... ce qui est peu réaliste

il faut poursuivre la réflexion un pas plus loin (encore !) et considérer désormais que seuls les tableaux partiels (possédant une taille logique indiquant le nombre d'éléments encodés) ont du sens

## c) TABLEAUX PARTIELS

Qu'est ce qu'un tableau partiel ?

Jusqu'à présent, nous avons considéré que c'était une variable-tableau physique (dont la taille physique est déterminée par une constante) à laquelle était associée une variable entière (jouant le rôle de taille logique)

mais si l'on reprend le problème de cotations et qu'on déclare ceci :

```
Constante coteMin = 0.0, coteMax = 20.0, nbCotes = 50
Type TETuCote = Enregistrement
    nom : Texte
    cote : Réel
    finEnr
Type TtbEtuCotes = Tableau[nbCotes] de TETuCote
Variable tbCotation1TL, tbCotation1TM : TtbEtuCotes # tableaux
    nbCotation1TL, nbCotation1TM : Entier # tailles logiques
```

on remarque qu'on est de nouveau 'coincé' dans la rédaction de l'API :

- la fonction lireEtuCotes ne sait pas renvoyer deux valeurs (le tableau et sa taille logique), donc il faut à nouveau avoir recours à une procédure (à laquelle on passerait comme paramètres par référence le tableau et sa variable-taille)

```
Procédure lireEtuCotes(ref tEC : TtbEtuCotes, ref nb : Entier) :
Variable indice : Entier # variable locale itérateur
    fini : Logique # état de la saisie (fin à la demande)
    nc : TETuCote # enregistrement local pour la saisie
Début
    nb ← 0 # tableau vide !!
    nc ← lireEtuCote() # première saisie
    fini ← (nc.nom = "****") # saisie terminée si nom étudiant = "****"
    tantQue non fini Faire
        nb ← nb + 1 # une cellule de plus
        tEC[nb] ← nc # mémoriser données enregistrement local
        nc ← lireEtuCote() # saisie suivante
        fini ← (nc.nom = "****") # saisie terminée ?
    finTQ
Fin
```

- la procédure afficherEtuCotes doit elle aussi accepter deux paramètres :

```
Procédure afficherEtuCotes(tEC : TtbEtuCotes, nb : Entier) : # 2 param!
Variable indice : Entier
Début
    Pour indice de 1 à nb Faire
        écrire(tEC[indice].nom, tEC[indice].cote)
    finPr
Fin
```

- sans compter l'alourdissement de la séquence principale :

```
Début # séquence principale
...
    lireEtuCotes(tbCotation1TL, nbCotation1TL) # 2 paramètres
    lireEtuCotes(tbCotation1TM, nbCotation1TM)

    afficherEtuCotes(tbCotation1TL, nbCotation1TL) # 2 paramètres
    afficherEtuCotes(tbCotation1TM, nbCotation1TM)
...
Fin
```

Et si on considérait que le tableau et sa variable-taille étaient les deux facettes complémentaires d'une seule et même structure ?

Tout naturellement (?), on va faire d'un tableau partiel un enregistrement !

Déclarons, déclarons ...

```

Constante coteMin = 0.0, coteMax = 20.0, nbCotes = 50
Type TEtuCote = Enregistrement
    nom : Texte
    cote : Réel
    finEnr
Type TtbEtuCotes = Tableau[nbCotes] de TEtuCote           # tableau physique
Type TtbpEtuCotes = Enregistrement                       # tableau partiel
    nbEtuCotes : Entier                                    # taille logique
    tbEtuCotes : TtbEtuCotes                             # tableau
    finEnr
Variable tbpCotation1TL, tbpCotation1TM : TtbpEtuCotes    # tableaux partiels

```

cette fois, on a un enregistrement qui contient un tableau d'enregistrements et un entier ! sans doute est-il nécessaire de refaire des exercices de nommage : pour rappel : les accesseurs s'utilisent de l'extérieur (ici le nom générique de l'enregistrement d'abord) vers l'intérieur (puis le nom des cellules du tableau ensuite, et enfin les champs de l'enregistrement constitutif des cellules ...)

ainsi pour la variable `tbpCotation1TL`,

- le nombre de cotations encodées a pour nom

```
tbpCotation1TL.nbEtuCotes
```

- le nom du cinquième étudiant a pour nom

```
tbpCotation1TL.tbEtuCotes[5].nom
```

- la valeur de la dernière cote (encodée) a pour nom :

```
tbpCotation1TL.tbEtuCotes[tbpCotation1TL.nbEtuCotes].cote
```

Puisqu'on a récupéré une structure unique (un tableau qui embarque avec lui sa taille logique, le tout constituant une valeur), les fonctions vont pouvoir faire leur retour définitif !

#### d) PETITE API BASIQUE POUR LA GESTION DE TABLEAUX PARTIELS

Un exemple concret de ce que l'on peut faire avec un tableau (basique) partiel : ajouter, modifier et supprimer des éléments devient envisageable; on devra cependant définir soigneusement le comportement et les contraintes des procédures et fonctions de l'API souhaitée ...

Cet exemple concerne un simple tableau partiel dont les cellules peuvent contenir chacune un Texte (p.ex. un mot), l'ADT correspond aux déclarations suivantes :

```

Constante maxMots = 50                                     # taille physique
Type TtbMots = Tableau[maxMots] de Texte                 # tableau physique
Type TtbpMots = Enregistrement                          # tableau partiel
    nbMots : Entier                                       # taille logique
    tbMots : TtbMots                                    # tableau
    finEnr
Variable tb1, tb2 : TtbpMots                            # variables tableaux partiels

```

- comme signalé précédemment, on se trouve parfois à hésiter entre la rédaction d'une procédure ou d'une fonction ... dans un cas comme celui-ci, la plupart du temps, on modifie le tableau (on change son contenu), dès lors une procédure paramétrée (avec un tableau-paramètre passé par référence) semble la plus adéquate

- initialiser (vider) un tableau de mots :

contrainte à prévoir : sur un tableau existant, demander confirmation puisque perte du contenu

Fonction init() : TtbpMots

Variable t : TtbpMots

Début

t.nbMots ← 0

renvoie(t)

Fin

*# invocation :*

tb2 ← init() *# nouveau*

Si tb1.nbMots ≠ 0 Alors *# existant*

écrire("sûr (o/n) ? ")

lire(réponse)

Si réponse = 'o' Alors tb1 ← init() finSi

finSi

Procédure init(ref t : TtbpMots) :

Début

t.nbMots ← 0

Fin

*# invocation :*

init(tb2) *# nouveau*

Si tb1.nbMots ≠ 0 Alors *# existant*

écrire("sûr (o/n) ? ")

lire(réponse)

Si réponse = 'O' Alors init(tb1) finSi

finSi

- ajouter un élément à une position précise : (mettre le mot "tantQue" dans la cellule 4) :

contraintes à prévoir : vérifier que la position spécifiée est possible, vérifier que le tableau n'est pas plein; le tableau partiel doit rester complet (pas de trou entre les positions 1 et nbMots), donc l'ajout à la position spécifiée n'est autorisé que si la position est entre 1 et nbMots; les éléments existants d'indice ≥ position doivent alors être reculés d'un cran vers la droite pour libérer la position d'insertion; si la position spécifiée est supérieure à nbMots, l'élément est inséré en position nbMots + 1

Procédure addMot(ref t : TtbpMots, m : Texte, p : Entier) :

*# ajouter le mot m dans le tableau t à l'indice p*

Variable indice : Entier

Début

SelonQue

p > maxMots : *# traitement erreur position incorrecte*

t.nbMots = maxMots : *# traitement erreur tableau rempli*

p entre 1 et t.nbMots : *# décalages avant insertion*

*Pour indice de t.nbMots à p par -1 Faire # décalages à droite*

t.tbMots[indice + 1] ← t.tbMots[indice]

finPr

t.tbMots[p] ← m *# insertion*

t.nbMots ← t.nbMots + 1 *# il y a un mot en plus !*

Sinon *# nouveau mot à la suite ...*

t.nbMots ← t.nbMots + 1 *# un mot de plus*

t.tbMots[t.nbMots] ← m *# insertion en dernier*

finSQ

Fin

*# invocation :*

addMot(tb2, "tantQue", 5) *# nouveau*

- supprimer un élément à une position précise : (enlever le mot de la cellule 7) :

contraintes à prévoir : vérifier que la position spécifiée est entre 1 et nbMots (ce qui inclut le test du tableau vide); le tableau partiel doit rester complet (pas de trou entre les positions 1 et nbMots), donc la suppression consiste à reculer les éléments existants d'indice  $\geq$  position d'un cran vers la droite pour 'remplir' le trou; ne pas oublier de décrémenter le nombre de mots

Procédure supprMot(ref t : TtbpMots, p : Entier) :

```
# supprimer le mot du tableau t à l'indice p
Variable indice : Entier
Début
  Si p > t.nbMots Alors
    # traitement erreur position incorrecte
  Sinon
    Pour indice de p à t.nbMots Faire # décalages à gauche
      t.tbMots[indice] ← t.tbMots[indice + 1]
    finPr
    t.nbMots ← t.nbMots - 1 # il y a un mot en moins !
  finSi
Fin

# invocation :
supprMot(tbl, 7) # enlever
```

- remplacer le contenu d'une cellule : (remplacer le mot de la cellule 8 par "Répéter")

contraintes à prévoir : la cellule doit contenir un mot, donc la position spécifiée doit être comprise entre 1 et nbMots

Procédure remplMot(ref t : TtbpMots, p : Entier, m : Texte) :

```
# remplacer la cellule d'indice p du tableau t par le mot m
Début
  Si non (p entre 1 et t.nbMots) Alors
    # traitement erreur position incorrecte
  Sinon
    t.tbMots[p] ← m
  finSi
Fin

# invocation :
remplMot(tbl, 7, "Pour") # remplacer
```

- lister le tableau

contraintes à prévoir : aucune

Procédure listerMots(t : TtbpMots) :

```
# afficher le contenu du tableau partiel t
Variable indice : Entier
Début
  Pour indice de 1 à t.nbMots Faire # simple itération
    écrire(t.tbMots[indice])
  finPr
Fin

# invocation :
listerMots(tbl) # afficher
```

e) EXERCICES

1. *Créer un ADT/API pour un tableau `tbEmp` qui contiendra les informations sur un maximum de 50 employés d'une entreprise (matricule, nom, salaire, étatCivile, dateNaissance, dateEntrée),*
    - *prévoir une procédure d'initialisation, des procédures et/ou fonctions pour l'encodage et les validations validation, une procédure d'affichage avec filtre sur le salaire (cfr ci-dessous)*
    - *encoder une dizaine d'employés puis afficher le nombre d'employés dont le salaire est compris entre 2,000 et 2,500 €*
    - *nb : les états civils valides sont ('C', 'M', 'D', 'V', 'S'); les salaires mensuels doivent être compris entre 1,500 et 3,500€; on vous permet de ne pas valider les années bissextiles !*
- 

2. *Si on souhaite sauvegarder momentanément les informations concernant des étudiants pour un traitement, nous avons besoin d'écrire une structure permettant d'enregistrer le matricule de l'étudiant, son nom, son sexe, son année de naissance, la cote (/20) de deux à sept interrogations, la moyenne obtenue et la mention 'ajourné' ou 'réussi' pour une moyenne supérieure ou égale à 10/20*

*On demande de définir cette structure d'étudiant ainsi que celle d'un tableau qui permet d'en stocker 80 maximum*

*Rédigez une API permettant*

- *d'encoder séparément les renseignements signalétique pour un étudiant (matricule, nom, sexe, année) à la demande*
  - *d'encoder avec validation – sur base de l'indice connu d'un étudiant dans le tableau – tout ou partie de ses cotes (p.ex. encoder 3 cotes pour l'étudiant 8)*
  - *de calculer la moyenne et la mention pour l'ensemble des étudiants*
  - *d'afficher l'identité et les résultats de l'ensemble des étudiants*
-





## 5.5. ALGORITHMES CLASSIQUES SUR LES STRUCTURES AVEC TABLEAUX

### a) TABLEAUX À CORRESPONDANCE DIRECTE

Il y a des tableaux dont l'exploitation est tellement immédiate que l'on oublie généralement d'en parler ... ; en deux mots :

ce sont des tableaux dont la taille est fixe et prédéterminée (pas besoin de déranger une taille logique) et pour lesquels il y a une correspondance directe et évidente entre l'indice et le contenu de la cellule à cet indice; dès lors, l'utilisation du tableau est d'une grande simplicité

Exemple :

```
Type TtbMois = Tableau[12] de Texte
Type tDate = Enregistrement
    jour : Entier
    mois : Entier
    année : Entier
finEnr
Variable tbMois : TtbMois
    dte : TDate

Procédure libMois :    # chargement
Début
    tbMois[1] ← "janvier"
    tbMois[2] ← "février"
    ...
    tbMois[12] ← "décembre"
Fin
```

Lorsqu'on aura une date à formater, au lieu d'écrire un long et coûteux SelonQue, il sera plus simple d'écrire (puisque'il est évident que la valeur du mois 3 permet d'accéder au contenu "mars" de la cellule d'indice 3)

```
écrire(dte.jour, ' ', tbMois[dte.mois], ' ', dte.année)
```

### b) RECHERCHE(S) DANS UN TABLEAU NON ORDONNÉ

Les structures de tableaux sont très utiles pour stocker en mémoire un ensemble de valeurs ...

Il est souvent demandé à de telles structures de répondre à une question fondamentale telle que : est-ce que *telle* valeur particulière se trouve dans le tableau et si oui à quelle position (indice) ? ...

... avec des variantes intéressantes : est-ce que *telle* valeur particulière se trouve dans le tableau et si oui à quelles positions (indices) ? ou encore : est-ce que *telle* valeur particulière se trouve dans le tableau et si oui quelle est sa dernière (sa plus haute) position (indice) ?

Exemples : sur l'ADT décrit précédemment

```
Constante coteMin = 0.0, coteMax = 20.0, nbCotes = 50
Type TEtuCote = Enregistrement
    nom : Texte
    cote : Réel
finEnr
Type TtbEtuCotes = Tableau[nbCotes] de TEtuCote    # ADT noms et cotes
Variable tbpCotation1TL : TtbEtuCotes
```

*quelle est la cote de Mr Dugenou ?*

*mademoiselle Duraton a-t-elle fait l'interro ?*

*quelqu'un a-t-il obtenu la cote de 20 ? qui l'a obtenue ?*

Cette fois, plus de correspondance directe, il va falloir chercher ... en itérant (déterminer d'abord l'indice, pour pouvoir ensuite exploiter le contenu ...)

On étudiera plus loin des algorithmes de recherche qui se basent sur la certitude que le tableau est préalablement ordonné (trié) ; si ce n'est pas le cas, il reste à explorer systématiquement (tout) le tableau (ces algorithmes sont appelés pour cette raison recherche séquentielle), l'arrêt plus ou moins rapide de cette exploration dépendant du type de question posée

- Recherche de la première occurrence d'une valeur : commençons avec une structure simple : un tableau partiel non typé et une procédure de recherche sans paramètre (tout se fait au moyen de variables globales)

```

Constante maxElem = 200                                # taille physique du tableau
Variable tb : tableau[maxElem] de Entier                # tableau d'entiers
    nbElem : Entier                                     # taille logique du tableau
    trouvé, fini : Logique                             # état de la recherche
    valeur : Entier                                    # ce que l'on cherche
    indice: Entier                                     # indice d'itérateur

Procédure chargerTableau : # à rédiger selon le problème posé
Début
    ...
Fin # on suppose ici que les nbElem premières cellules de tb ont été encodées

Procédure rechercheTableau1 :
Début
    trouvé ← faux                                     # pessimisme !
    indice ← 1                                       # initialisation itérateur
    fini ← (indice > nbElem)                         # ne pas oublier un tableau vide !!!
    tantQue non (trouvé ou fini)                    # précaution avec 2 conditions d'arrêt
        Si tb[indice] = valeur Alors
            trouvé ← vrai                             # yessssssssssss !
        Sinon
            indice ← indice + 1                       # no, continue !
            fini ← (indice > nbElem)                  # possible de continuer ?
        finSi
    finTQ
Fin

Début # séquence principale
    chargerTableau
    écrire("quelle valeur cherchez-vous ? ")
    lire(valeur)
    rechercheTableau1
    Si trouvé
        Alors écrire("trouvé ", valeur, " à l'indice ", indice)
        Sinon écrire("pas trouvé ", valeur)
    finSi
Fin.
```

*Remarques :*

1. Une variante sans variable logique de cet algorithme existe : il suffit de comparer la valeur recherchée et l'élément désigné par l'indice en sortie de la procédure de recherche (mais une fois encore, l'utilisation d'une variable logique permet de s'affranchir des données dans la séquence principale) :

```

Début
    chargerTableau
    lire(valeur)
    rechercheTableau2
    Si (tb[indice]) = valeur
        Alors écrire("trouvé ", valeur, " à l'indice ", indice)
        Sinon écrire("pas trouvé ", valeur)
    finSi
Fin.
```

2. Si l'on souhaite rédiger cette recherche avec les 'belles' boucles 'Répéter' ou 'tantQue', il faut veiller à la bonne gestion de l'indice, surtout au moment de sortir de la boucle quand on a trouvé ! :

Au début du chapitre consacré aux Tableaux, au moment de décrire les itérateurs basiques, deux versions équivalentes de chaque type boucle (du point de vue de l'initialisation, de l'incréméntation de l'indice et de l'utilisation de ce dernier) ont été données ; dans le cas de la recherche, l'une – du fait de la position de l'incréméntation – nécessite un sinon supplémentaire, l'autre pas

Dans le cas présent, dans la mesure où un tableau partiel peut être vide, il faut oublier le 'Répéter' dépourvu de principe de précaution !

*version 'plus lourde'*

```
Procédure rechercheTableau2 :
Début
    trouvé ← faux
    indice ← 1
    fini ← (indice > nbElem)           # ne pas oublier un tableau vide !!!
    tantQue non (fini ou trouvé)
        Si tb[indice] = valeur
            Alors trouvé ← vrai
            Sinon indice ← indice + 1
                fini ← (indice > nbElem)
        finSi
    finTQ
Fin
```

*version 'plus légère' (remarquer l'initialisation de l'indice d'itérateur)*

```
Procédure rechercheTableau3 :
Début
    trouvé ← faux
    indice ← 0
    fini ← (indice = nbElem)           # ne pas oublier un tableau vide !!!
    tantQue non (fini ou trouvé)
        indice ← indice + 1
        trouvé ← (tb[indice] = valeur)
        fini ← (indice = nbElem)
    finTQ
Fin
```

Tout cela est intéressant et instructif, mais manque un peu de consistance et de cohérence :

- l'utilisation de variables globales empêche la réutilisation de la procédure pour une recherche similaire dans un autre tableau
- cette réutilisation demande cependant un passage de paramètre(s) typé(s) ... (ici, il y a absence de types abstraits et donc la procédure ne peut prétendre à être une composante d'API)
- et surtout : pourquoi une procédure de recherche ?? qui en plus détermine et un état global (trouvé) et une variable globale (indice) nécessaire à la séquence principale ... une fonction de recherche ne serait-elle pas mieux adaptée ?

- Recherche de la première occurrence d'une valeur (suite) : travaillons à présent dans l'esprit ADT/API; outre des déclarations plus adaptées, nous allons écrire une fonction de recherche chargée de retourner une valeur : l'indice de la cellule, correspondant au succès de la recherche ... et en cas d'échec ? la fonction retournera tout simplement la valeur 0 (qui ne correspond à aucun indice valable)<sup>69</sup>

```

Constante maxElem = 200                                # taille physique du tableau
Type TtbEnt = Tableau[maxElem] de Entier                # ADT tableau d'entiers
Type TtbpEnt = Enregistrement                           # ADT tableau partiel
    tbEnt : TtbEnt                                     # tableau
    nbElem : Entier                                    # taille logique du tableau
finEnr
Variable tb1, tb2 : TtbpEnt                             # 2 variables "tableau partiel"
    valeur : Entier                                    # ce que l'on cherche
    indiceRetour : Entier                              # résultat de la recherche (0 ou indice)

Procédure chargerTableau(ref t : TtbpEnt) :             # selon le problème posé
Début
    ...                                                # un itérateur, peu importe lequel, contenant :
    t.nbElem ← t.nbElem + 1
    t.tbEnt[t.nbElem] ← ...
    ...
Fin # on suppose ici que les nbElem premières cellules de t ont été encodées

Fonction chercherTableau(t : TtbEnt, v : Entier) : Entier :
Variable indice : Entier; trouvé, fini : Logique
Début
    trouvé ← faux
    indice ← 0
    fini ← (indice = t.nbElem)                        # ne pas oublier un tableau vide !!!
    tantQue non (fini ou trouvé)
        indice ← indice + 1
        trouvé ← (t.tbEnt[indice] = valeur)
        fini ← (indice = t.nbElem)
    finTQ
    Si trouvé
        Alors renvoie(indice)
        Sinon renvoie(0)
    finSi
Fin

Début # séquence principale
    chargerTableau(tb1)
    chargerTableau(tb2)
    écrire("quelle valeur cherchez-vous ? ")
    lire(valeur)
    indiceRetour ← chercherTableau(tb1, valeur)
    Si indiceRetour ≠ 0 Alors
        écrire("trouvé dans le premier tableau à l'indice ", indiceRetour)
    Sinon
        indiceRetour ← chercherTableau(tb2, valeur)
        Si indiceRetour ≠ 0 Alors
            écrire("trouvé dans le second tableau à l'indice ", indiceRetour)
        Sinon
            écrire("valeur introuvable !")
        finSi
    finSi
Fin.

```

<sup>69</sup> dans le cas du langage C qui commence tous ses indices à 0, il est habituel de renvoyer -1 en cas d'insuccès

- Recherche de la dernière occurrence d'une valeur :

Il suffit de reprendre l'algorithme de recherche de la première occurrence et d'enlever la sortie forcée de la boucle (de la sorte, on parcourt tout le tableau) MAIS il faut bien entendu veiller à conserver la mémoire du dernier indice de recherche fructueuse ; cet indice pourra jouer le double rôle de variable logique et de position (s'il est  $\neq 0$  : succès et position) !

Avec les mêmes déclarations et le même algorithme principal qu'à la page précédente ...

```
Fonction chercherTableau(t : TtbEnt, v : Entier) : Entier :
Variable indice, indiceRetour : Entier
Début
    indiceRetour ← 0                # pour la facilité de la fin ...
    Pour indice de 1 à t.nbElem Faire
        Si t.tbEnt[indice] = valeur Alors indiceRetour ← indice finSi
    finPr
    renvoie(indiceRetour)
Fin
```

- Recherche de toutes les occurrences d'une valeur :

Le principe illustré ici est le suivant : on va conserver au sein d'un nouveau tableau les différents indices des éléments dont la valeur est égale à celle recherchée.

(l'algorithme utilise pour le tableau de résultats l'idée de l'ajout d'éléments dans un tableau partiel étudiée précédemment)

```
Constante maxElem = 200                # taille physique du tableau
Type TtbEnt = Tableau[maxElem] de Entier # ADT tableau d'entiers
Type TtbpEnt = Enregistrement          # ADT tableau partiel
    tbEnt : TtbEnt                      # tableau
    nbElem : Entier                     # taille logique du tableau
    finEnr
Variable tb, tbIndices : TtbpEnt        # 2 variables "tableau partiel"
    valeur : Entier                     # ce que l'on cherche
```

```
Fonction chercherToutes(t : TtbEnt, v : Entier) : TtbpEnt :
Variable tbInd : TtbpEnt                # tableau avec les indices trouvés
    indice : Entier                      # indice d'itérateur
Début
    tbInd.nbElem ← 0                    # encore rien trouvé !
    Pour indice de 1 à t.nbElem Faire
        Si t.tbEnt[indice] = valeur Alors # si trouvé
            tbInd.nbElem ← tbInd.nbElem + 1 # incrémenter nombre trouvés
            tbInd.tbEnt[tbInd.nbElem] ← indice # et mémoriser indice trouvé
        finSi
    finPr
    renvoie(tbInd)
Fin
```

```
Procédure afficherRésultats(t : TtbEnt) :
Variable indice : Entier
Début
```

```
    Si t.nbElem > 0 Alors
        écrire("liste des indices : ")
        Pour indice de 1 à t.nbElem Faire
            écrire("trouvé à l'indice ", t.tbEnt[indice])
        finPr
    Sinon
        écrire("pas trouvé ")
    finSi
Fin
```

```
Début
    chargerTableau; lire(valeur)        # que cherche-t-on ?
    tbIndices ← chercherToutes(tb, valeur) # recherche ...
    afficherRésultats(tbIndices)         # ... affichage de ce qu'on a trouvé
Fin.
```

- une élégante alternative : la méthode "de la sentinelle"

on pourrait reprocher à l'algorithme classique de recherche une certaine lourdeur due à l'utilisation de deux états

```

Fonction chercherTableau(t : TtbEnt, v : Entier) : Entier :
Variable indice : Entier; trouvé, fini : Logique
Début
    indice ← 0
    trouvé ← faux
    fini ← (indice = t.nbElem)      # ne pas oublier un tableau vide !!!
    tantQue non (fini OU trouvé)
        indice ← indice + 1
        trouvé ← (t.tbEnt[indice] = valeur)
        fini ← (indice = t.nbElem)
    finTQ
    Si trouvé
        Alors renvoie(indice)
        Sinon renvoie(0)
    finSi
Fin

```

il existe donc un algorithme plus 'léger' qui ne demande qu'un seul état (trouvé); il est basé sur le principe suivant : on place la valeur à rechercher dans le tableau, derrière le contenu existant (une case derrière la dernière, donc), puis on itère le tableau à la recherche de cette valeur : on la trouvera forcément ! si c'est à l'endroit où on l'a placée, c'est l'échec; si on la trouve avant cet endroit c'est réussi !

```

Fonction chercherTableau(t : TtbEnt, v : Entier) : Entier :
Variable indice : Entier; trouvé : Logique
Début
    t.tbEnt[t.nbElem + 1] ← v
    indice ← 0
    trouvé ← faux
    tantQue non trouvé
        indice ← indice + 1
        trouvé ← (t.tbEnt[indice] = valeur)
    finTQ
    Si trouvé
        Alors renvoie(indice)
        Sinon renvoie(0)
    finSi
Fin

```

pour les programmeurs décidément allergiques à l'utilisation des états logiques, une dernière 'simplification' est même possible !

```

Fonction chercherTableau(t : TtbEnt, v : Entier) : Entier :
Variable indice : Entier
Début
    t.tbEnt[t.nbElem + 1] ← v
    indice ← 1
    tantQue t.tbEnt[indice] ≠ valeur
        indice ← indice + 1
    finTQ
    Si indice < t.nbElem + 1
        Alors renvoie(indice)
        Sinon renvoie(0)
    finSi
Fin

```

la seule précaution à prendre est de dimensionner le tableau d'une case supplémentaire et/ou bien d'arrêter son chargement une case avant la fin (un tableau de N cases est rempli si N-1 cases sont occupées)







Remarques finales :

- ° cet algorithme de tri va nécessiter un grand nombre de d'échanges d'éléments ; si le tableau est de taille N (N éléments), il va nécessiter N-1 boucles principales dans le cas où le dernier élément doit être placé en premier ; le nombre de boucles internes maximal est donc de l'ordre de  $(N-1)^2$
- ° si le tableau (de taille N éléments) est ordonné au départ, un seul parcours suffit (et aucun échange n'est opéré)
- ° si le tableau (de taille N éléments) est ordonné 'à l'envers' (il est trié décroissant alors qu'on souhaite un tri croissant, p.ex.), il faudra  $(N-1)^2$  parcours pour l'ordonner

Question :

- ° dans la mesure où à chaque parcours, on a la certitude de faire 'monter' chaque fois la plus grande valeur (suivante), on peut sans doute améliorer l'algorithme en ne faisant pas systématiquement (comme c'est le cas ci-dessus) une boucle de 1 à t.nbElem - 1, mais au deuxième parcours une boucle de 1 à t.nbElem - 2 suffit, au troisième parcours de 1 à t.nbElem - 3, etc ...

l'écriture de cette version (un rien) plus économique et laissée aux bons soins du lecteur

Un dernier mot : l'algorithme écrit ci-dessus (décomposé en deux procédures) présente un inconvénient : il nécessite une variable logique que l'on doit déclarer globale car elle est partagée par les deux procédures; compte tenu de la 'petitesse' de ces procédures, on les fusionnera avantageusement (à remarquer qu'il s'agit bien de deux boucles imbriquées : la première, un Répéter, prend en charge le but à atteindre, la seconde, un Pour, itère à chaque fois tout le tableau et réalise les échanges éventuels) :

```

Procédure triBulle(ref t : TtbEnt) :
Variable : ech : Entier                # variable pour l'échange
         indice: Entier                # indice d'itérateur
         tableauTrié : Logique         # état final espéré
Début
  Répéter
    tableauTrié ← vrai
    Pour indice de 1 à t.nbElem - 1 Faire      # !! borne supérieure !
      Si t.tbEnt[indice] > t.tbEnt[indice + 1] Alors # si non respect ...
        ech ← t.tbEnt[indice]                  # ... échange
        t.tbEnt[indice] ← t.tbEnt[indice + 1]
        t.tbEnt[indice + 1] ← ech
        tableauTrié ← faux                    # potentiellement pas trié !
      finSi
    finPr
  Jusque tableauTrié
Fin

```

b) LE TRI 'PAR SÉLECTION'

*repreons le même tableau (avec les mêmes déclarations et les mêmes données) :*

[illegible]

On va également parcourir le tableau avec un itérateur mais cette fois, à chaque parcours, on va rechercher la plus petite valeur et la mettre directement 'à sa place' par échange;

ici, au premier parcours -5 sera placé à la case d'indice 1 (on échange 3 et -5), au parcours suivant 0 sera placé à l'indice 2 (on échangera 6 et 0), etc ...

```

Constante maxElem = 200                                # taille physique du tableau
Type TtbEnt = Tableau[maxElem] de Entier                # ADT tableau d'entiers
Type TtbpEnt = Enregistrement                           # ADT tableau partiel
    tbEnt : TtbEnt                                     # tableau
    nbElem : Entier                                    # taille logique du tableau
finEnr
Variable tb1, tb2 : TtbpEnt                             # 2 variables "tableau partiel" à trier

Procédure echangeMin(ref t : TtbEnt) :
# une itération = tranfert du minimum et début de tableau
Variable ech : Entier                                  # variable pour faire l'échange
    indice: Entier                                     # indice d'itérateur
    indMin : Entier                                    # position (indice) du minimum
Début
    indMin ← 1                                          # initialisation de la position du minimum
    Pour indice de 2 à t.nbElem Faire                 # boucle de recherche !! bornes !!
        Si t.tbEnt[indice] < t.tbEnt[indMin] Alors    # si plus petit que le minimum
            indMin ← indice                            # mémoriser sa position
        finSi
    finPr
    ech ← t.tbEnt[1]                                   # procéder à l'échange
    t.tbEnt[1] ← t.tbEnt[indMin]
    t.tbEnt[indMin] ← ech
Fin

```

[illegible]

Pour ordonner l'ensemble du tableau, donc le trier, il suffit d'enchaîner des parcours en 'remontant' chaque fois la borne inférieure d'une unité ; cela s'écrit aisément avec deux boucles imbriquées ; attention toutefois à bien gérer pour chacune la borne inférieure et la borne supérieure !

```

Procédure triSélection(ref t : TtbEnt) :
Variable ech : Entier                                # variable pour faire l'échange
    indMin : Entier                                  # position (indice) du minimum
    indicel, indice2 : Entier                        # indices d'itérateurs

Début
    Pour indicel de 1 à t.nbElem - 1 Faire # itérateur externe
        indMin ← indicel                        # minimum = élément de borne inférieure
        Pour indice2 de indicel + 1 à t.nbElem Faire # itérateur interne
            Si t.tbEnt[indice2] < t.tbEnt[indMin] Alors # un nouveau minimum ?
                indMin ← indice2                # si oui, conserver sa position
            finSi
        finPr
        Si indMin ≠ indicel Alors                # pas la peine de s'échanger soi-même
            ech ← t.tbEnt[indicel]                # échanger
            t.tbEnt[indicel] ← t.tbEnt[indMin]
            t.tbEnt[indMin] ← ech
        finSi
    finPr
Fin

```

```

Début
  chargerTableau
  triSélection
Fin.

```

*Une trace de l'exécution de cet algorithme sur le tableau-exemple figure ci-contre :*

3	6	5	11	9	2	99	12	32	0	14	-5	1	87
-5	6	5	11	9	2	99	12	32	0	14	3	1	87
-5	0	5	11	9	2	99	12	32	6	14	3	1	87
-5	0	1	11	9	2	99	12	32	6	14	3	5	87
-5	0	1	2	9	11	99	12	32	6	14	3	5	87
-5	0	1	2	3	11	99	12	32	6	14	9	5	87
-5	0	1	2	3	5	99	12	32	6	14	9	11	87
-5	0	1	2	3	5	6	12	32	99	14	9	11	87
-5	0	1	2	3	5	6	9	32	99	14	12	11	87
-5	0	1	2	3	5	6	9	11	99	14	12	32	87
-5	0	1	2	3	5	6	9	11	12	14	99	32	87
-5	0	1	2	3	5	6	9	11	12	14	99	32	87
-5	0	1	2	3	5	6	9	11	12	14	32	99	87
-5	0	1	2	3	5	6	9	11	12	14	32	87	99

Remarques finales :

- ° on peut bien entendu baser l'algorithme sur la recherche du maximum (tri décroissant) que l'on placerait alors en fin de tableau (version laissée au lecteur)
- ° chaque parcours met un élément en position définitive (ici le plus petit), l'autre par contre est a priori mal placé ; par contre, ici, contrairement au tri-bulle, aucun autre échange n'est utile. Un élément qui a été bien placé ne sera plus testé par la suite. Le nombre de boucles internes est environ  $N(N-1)/2$ , ce qui est meilleur que le tri bulle, mais toujours de l'ordre de  $N^2$

c) VARIATIONS SUR LE THÈME DU TRI

- Problèmes multi-tableaux : clé de tri simple

À une époque où les structures (typées) d'Enregistrement n'étaient pas (encore) disponibles, les problèmes résolus à l'aide de tableaux en nécessitaient souvent plusieurs ... ce qui alourdissait la rédaction des algorithmes de tri

Essayons cela "à l'ancienne" (des variables-tableaux globales non typées manipulées directement par les procédures) sur un thème abordé précédemment : introduire le nom et la cote/20 pour un ensemble de N étudiants encodés 'en vrac' (nom + cote, bien sûr, mais dans un ordre quelconque des noms) : on devra déclarer (les tableaux sont considérés partiels)

```

Constante maxEtu = 50                                # taille physique
Variable tbNoms : tableau[maxEtu] de Texte          # tableau de noms
          tbCotes : tableau[maxEtu] de Réel         # tableau de cotes
          nbEtu : Entier                               # taille logique

```



## ▪ Problèmes multi-tableaux & clé composite : structures de données typées

Voyons si les structures mixtes typées "de haut niveau" vont apporter une aide significative ... en tous cas, elles vont permettre, quelle que soit la complexité de la structure

- d'abstraire le tri, puis que le passage de paramètres (typés) va être possible
- de simplifier définitivement l'échange, via une seule variable-cellule de travail

*l'algorithme ci-dessous est également un exercice de lecture de nommage de composants de structures mixtes ...*

```

Constante maxEtu = 50                                # taille physique
Type TEtuCote = Enregistrement                       # enregistrement etu + cote
    nom : Texte                                       # composant etu
    cote : Réel                                       # composant cote
finEnr
Type TtbEtuCotes = Tableau[maxEtu] de TEtuCote      # tableau physique
Type TtbpEtuCotes = Enregistrement                  # tableau partiel
    nbEtu : Entier                                    # taille logique
    tbEtuCotes : TtbEtuCotes                         # tableau
finEnr
Variable tbpCotation : TtbpEtuCotes                # variable tableau partiel

Procédure triSélection(ref t : TtbpEtuCotes) :
    Variable ind1, ind2, indMin                       # itérateurs et position du minimum
    ec : TEtuCote                                     # enregistrement pour échange

    Début
        Pour ind1 de 1 à t.nbEtu - 1 Faire
            indMin ← ind1
            Pour ind2 de ind1 + 1 à t.nbEtu Faire
                Si t.tbEtuCotes[ind2].cote < t.tbEtuCotes[indMin].cote Alors
                    indMin ← ind2
                Sinon
                    Si t.tbEtuCotes[ind2].cote = t.tbEtuCotes[indMin].cote et
                       t.tbEtuCotes[ind2].nom < t.tbEtuCotes[indMin].nom Alors
                        indMin ← ind2
                    finSi
                finSi
            finPr
            Si indMin ≠ ind1 Alors                    # pas la peine de s'échanger soi-même
                ec ← tbEtuCotes[ind1]                # échanger les cellules
                t.tbEtuCotes[ind1] ← t.tbEtuCotes[indMin]
                t.tbEtuCotes[indMin] ← ec
            finSi
        finPr
    Fin

```

NB : avec le sucre syntaxique Avec ... finAv, on peut économiser (un peu) sur la rédaction des comparaisons, p.ex. en privilégiant la cellule 'minimum'

```

Pour ind1 de 1 à t.nbEtu - 1 Faire
    indMin ← ind1
    Avec t.tbEtuCotes[indMin] Faire
        Pour ind2 de ind1 + 1 à t.nbEtu Faire
            Si (t.tbEtuCotes[ind2].cote < cote) OU
               (t.tbEtuCotes[ind2].cote = cote ET t.tbEtuCotes[ind2].nom < nom)
                Alors indMin ← ind2
            finSi
        finPr
    finAv
finPr

```

Ci-dessus, on a 'compacté' au maximum (grâce aux expressions logiques) la détection de la cellule 'minimale'; une technique plus générale (l'écriture d'un "comparateur") sera étudiée au second semestre dans le cadre du cours d'Organisation et Structure des Données ...

un dernier mot sur le tri : quel que soit l'algorithme retenu, quelle que soit la complexité des cellules (d'un simple nombre à un enregistrement complexe), on procède toujours à un échange (swap en anglais) de cellules et cet échange demande toujours une variable de travail (du même type que les cellules à échanger) et trois instructions d'affectation

dans un souci de simplification, on peut faire de l'échange une procédure qui recevra la référence des deux cellules à échanger et déclarera localement la variable d'échange :

Procédure **échanger**(**ref** c1 : **TEtuCote**, **ref** c2 : **TEtuCote**) :

Variable c : **TEtuCote** *# enregistrement pour échange*

Début

c ← c1

c1 ← c2

c2 ← c

Fin

Procédure **triSélection**(**ref** t : **TtbpEtuCotes**) :

Variable ind1, ind2, indMin *# itérateurs et position du minimum*

Début

Pour ind1 de 1 à t.nbEtu - 1 Faire

indMin ← ind1

Avec t.tbEtuCotes[indMin] Faire

Pour ind2 de ind1 + 1 à t.nbEtu Faire

Si (t.tbEtuCotes[ind2].cote < cote) OU

(t.tbEtuCotes[ind2].cote = cote ET t.tbEtuCotes[ind2].nom < nom)

Alors indMin ← ind2

finSi

finPr

finAv

Si indMin ≠ ind1 Alors

**échanger**(t.tbEtuCotes[ind1], t.tbEtuCotes[indMin])

finSi

finPr

Fin

bien entendu, cette procédure est très dépendante des types de données des cellules et il faut en écrire une spécifique à chaque fois; mais grâce au principe de surcharge évoqué au chapitre consacré aux procédures et aux fonctions, chacune des versions (à conserver p.ex. dans une librairie) portera le même nom, la signature levant l'ambiguïté

Procédure **échanger**(**ref** n1 : **Entier**, **ref** n2 : **Entier**) :

Variable n : **Entier**

Début

n ← n1

n1 ← n2

n2 ← n

Fin

Procédure **échanger**(**ref** t1 : **Texte**, **ref** t2 : **Texte**) :

Variable t : **Texte**

Début

t ← t1

t1 ← t2

t2 ← t

Fin

etc...

d) RECHERCHE DANS UN TABLEAU ORDONNÉ (TRIÉ)a) RECHERCHE SÉQUENTIELLE

Dès lors que le tableau est ordonné (ici supposé en ordre croissant), la recherche séquentielle de la première valeur ne nécessite plus le parcours de tout le tableau : rencontrer un élément de valeur supérieure à celle cherchée permet de s'arrêter en concluant à l'échec ; cet algorithme a donc trois conditions d'arrêt : la réussite, l'échec et bien entendu avoir atteint la fin du tableau

Par rapport à ce que nous avons rédigé précédemment pour la recherche séquentielle dans un tableau partiel non trié, la procédure de recherche ci-dessous est très peu modifiée, on rassemble sous un état commun (fini) les deux conditions d'échec ... et le tour est joué !

```

Constante maxElem = 200                                # taille physique du tableau
Type TtbEnt = Tableau[maxElem] de Entier                # ADT tableau d'entiers
Type TtbpEnt = Enregistrement                           # ADT tableau partiel
    tbEnt : TtbEnt                                     # tableau
    nbElem : Entier                                    # taille logique du tableau
    finEnr
Variable tb1, tb2 : TtbpEnt                             # 2 variables "tableau partiel"
    valeur : Entier                                     # ce que l'on cherche
    indiceRetour : Entier                               # résultat de la recherche (0 ou indice)

```

Procédure chargerTableau(ref t : TtbpEnt) :

Début

...

Fin # on suppose ici que les nbElem premières cellules de t ont été encodées

Procédure trierTableau(ref t : TtbpEnt) :

Début

...

Fin # on suppose ici que t est trié en ordre croissant

Fonction chercherTableau(t : TtbEnt, v : Entier) : Entier :

Variable indice : Entier; trouvé, fini : Logique

Début

trouvé ← faux

indice ← 0

**fini** ← (indice = t.nbElem) # ne pas oublier un tableau vide !!!

tantQue non (fini ou trouvé)

    indice ← indice + 1

    trouvé ← (t.tbEnt[indice] = valeur)

**fini** ← (t.tbEnt[indice] > valeur OU indice = t.nbElem)

finTQ

**Si** trouvé

    Alors renvoie(indice)

    Sinon renvoie(0)

finSi

Fin

Début # séquence principale

    chargerTableau(tb1)

    trierTableau(tb1)

    écrire("quelle valeur cherchez-vous ? ")

    lire(valeur)

    indiceRetour ← chercherTableau(tb1, valeur)

**Si** indiceRetour ≠ 0 **Alors**

        écrire("trouvé dans le premier tableau à l'indice ", indiceRetour)

**Sinon**

        écrire("valeur introuvable !")

    finSi

Fin.



b) RECHERCHE DICHOTOMIQUE

Qui n'a pas joué à 'plus grand / plus petit' ?

Il s'agit de deviner au plus vite le nombre que quelqu'un d'autre a en tête (dans un intervalle donné, disons de 1 à 1000).

- ° Le 'mauvais' joueur demandera : *est-ce 1 ? , est-ce 2 ? , est-ce 3 ? ...* et ainsi de suite jusqu'à la 'victoire' ; il utilise en fait une recherche séquentielle sur l'intervalle donné.

*Le coût de cette recherche est facile à déterminer : avec de la chance on trouve du premier coup (si le nombre à deviner est 1), avec un maximum de malchance, il faut s'y reprendre à 1000 fois (si le nombre à deviner est 1000) ; en moyenne donc, le coût (en nombre de comparaisons) est  $N/2$  ( $N$  étant le nombre de valeurs de l'intervalle)*

- ° Le 'bon' joueur adoptera la stratégie suivante : puisque l'intervalle est naturellement ordonné ( $1 < 2 < 3 < \dots < 1000$ ) : il essaie 'au centre' : *est-ce 500 ?* Trois possibilités de réponse : bravo, plus grand, plus petit !

- 'Bravo', il a gagné !
- 'Plus grand' permet d'éliminer la moitié inférieure du domaine (on cherche désormais entre 501 et 1000), et on recommencer à chercher en 'tapant' au milieu
- idem avec 'Plus petit' (on cherche désormais entre 1 et 499) et on recommencer à chercher en 'tapant' au milieu

*Comme on divise à chaque fois un (sous-)intervalle en deux, cette technique de recherche est appelée dichotomie (coupage en deux) et il est assez simple de montrer que le coût moyen (en nombre de comparaisons à effectuer) de la recherche est au pire  $\log_2 N + 1$  ( $N$  le nombre de valeurs de l'intervalle de départ) ; le petit tableau comparatif de la recherche séquentielle et de la recherche dichotomique ci-dessous montre clairement le bénéfice (plus l'intervalle est vaste)*

$N$	Coût séquentiel ( $N/2$ )	Coût dichotomie ( $\log_2 N + 1$ )
10	5	4
100	50	8
1,000	500	11
10,000	5,000	14
100,000	50,000	16
1,000,000	500,000	21
1,000,000,000	500,000,000	31

- ° Il est donc très intéressant de donner une version algorithmique de ce type de recherche sur les tableaux triés (déterminer le 'centre' et ensuite ne conserver que la 'bonne' moitié du tableau n'est rien d'autre qu'une manipulation correcte des indices, mais le fait que le nombre de cases soit pair ou impair pourrait être considéré comme un petit souci pour déterminer la 'moitié')
- ° Il est à noter encore que :
  - contrairement au jeu, la recherche n'est pas nécessairement fructueuse (la valeur recherchée n'est pas nécessairement dans le tableau) et qu'il va falloir apporter un soin tout particulier à bien rédiger la condition d'arrêt de la recherche
  - pour de 'petits' tableaux, la recherche infructueuse coûtera en moyenne plus cher par dichotomie que par séquence (pour la dichotomie, la recherche infructueuse coûte toujours  $\log_2 N + 1$  comparaisons)

Comme pour les tris, il existe différentes versions de cet algorithme, nous n'en retiendrons qu'une seule ; mais avant toute chose, essayons manuellement :

- Soit le tableau partiel  $t$  suivant (les cases sont surmontées de leur indice pour une meilleure compréhension) : 84 est-il dans le tableau ?

1	2	3	4	5	6	7	8	9	10	11	12	13	14						
2	5	9	11	19	23	42	50	55	71	84	92	99	125						

nous allons utiliser trois variables pour borner l'espace de recherche :  $bGauche$ ,  $bDroite$  et  $milieu$

prenons comme départ  $bGauche \leftarrow bInf$ ,  $bDroite \leftarrow bSup$  et  $milieu \leftarrow (bGauche + bDroite) \div 2$

comme  $bInf = 1$  et  $bSup = 14$ , alors  $bGauche=1$ ,  $bDroite=14$  et  $milieu = 7$

$t[milieu] = t[7] = 42$  : plus petit que 84 ! on peut donc éliminer le (sous-)tableau d'indices 1 à 7 et recommencer le processus sur le (sous-)tableau d'indices 8 à 14

1	2	3	4	5	6	7	8	9	10	11	12	13	14						
2	5	9	11	19	23	42	50	55	71	84	92	99	125						

Pour ce faire, prenons une nouvelle borne gauche en posant  $bGauche \leftarrow milieu + 1$  ; cela donne  $bGauche=8$ ,  $bDroite=14$  et un nouveau  $milieu = 11$

$t[milieu] = t[11] = 84$  ! trouvé en 2 coups !

- Soit le tableau partiel  $t$  suivant : 19 est-il dans le tableau ?

1	2	3	4	5	6	7	8	9	10	11	12	13	14						
2	5	9	11	19	23	42	50	55	71	84	92	99	125						

prenons comme départ  $bGauche \leftarrow bInf$ ,  $bDroite \leftarrow bSup$  et  $milieu \leftarrow (bGauche + bDroite) \div 2$

comme  $bInf = 1$  et  $bSup = 14$ , donc  $bGauche=1$ ,  $bDroite=14$  et  $milieu = 7$

$t[milieu] = t[7] = 42$  : plus grand que 19 ! on peut donc éliminer le (sous-)tableau d'indices 7 à 14 et recommencer le processus sur le (sous-)tableau d'indices 1 à 6

1	2	3	4	5	6	7	8	9	10	11	12	13	14						
2	5	9	11	19	23	42	50	55	71	84	92	99	125						

pour ce faire, prenons une nouvelle borne droite en posant  $bDroite \leftarrow milieu - 1$  ; cela donne  $bGauche=1$ ,  $bDroite=6$  et un nouveau  $milieu = 3$

$t[milieu] = t[3] = 9$  ! plus petit que 19 ; on élimine un autre morceau de tableau et on continue

1	2	3	4	5	6	7	8	9	10	11	12	13	14						
2	5	9	11	19	23	42	50	55	71	84	92	99	125						

prenons une nouvelle borne gauche en posant  $bGauche \leftarrow milieu + 1$  ; cela donne  $bGauche=4$ ,  $bDroite=6$  et  $milieu = 5$

$t[milieu] = t[5] = 19$  ! trouvé !

- Et si l'on avait cherché 20 ?

comme  $t[\text{milieu}] = t[5] = 19$  ! plus petit que 20,

on aurait posé  $b\text{Gauche} \leftarrow \text{milieu} + 1$ , ce qui aurait donné  $b\text{Gauche}=6$ ,  $b\text{Droite}=6$  et  $\text{milieu} = 6$  ;  $t[\text{milieu}] = t[6] = 23$ , plus grand que 19,

on aurait posé  $b\text{Droite} \leftarrow \text{milieu} - 1$ , ce qui aurait donné  $b\text{Gauche}=6$ ,  $b\text{Droite}=5$

à ce stade,  **$b\text{Gauche} > b\text{Droite}$  : condition d'arrêt et de recherche infructueuse !**

Il n'y a plus qu'à exprimer telle quelle cette logique de manière algorithmique :

```

Constante maxElem = 200                                # taille physique du tableau
Type TtbEnt = Tableau[maxElem] de Entier                # ADT tableau d'entiers
Type TtbpEnt = Enregistrement                          # ADT tableau partiel
    tbEnt : TtbEnt                                     # tableau
    nbElem : Entier                                    # taille logique du tableau
finEnr
Variable tb1, tb2 : TtbpEnt                            # 2 variables "tableau partiel"
    valeur : Entier                                    # ce que l'on cherche
    indiceRetour : Entier                             # résultat de la recherche (0 ou indice)

Procédure chargerTableau(ref t : TtbpEnt) :
Début
    ...
Fin    # on suppose ici que les nbElem premières cellules de t ont été encodées

Procédure trierTableau(ref t : TtbpEnt) :
Début
    ...
Fin    # on suppose ici que t est trié en ordre croissant

Fonction rechercheDichotomique(t : TtbpEnt, v : Entier) : Entier
Variable bGauche, bDroite, milieu : Entier             # les bornes de recherche
    fini, trouvé : Logique
Début
    bGauche ← 1                                         # init gauche
    bDroite ← t.nbElem                                 # init droite
    trouvé ← faux                                       # pas encore trouvé
    fini ← (bGauche > bDroite)                         # vrai si tableau vide, faux sinon
    tantQue non (fini ou trouvé) Faire                 # deux conditions d'arrêt
        milieu ← (bGauche + bDroite) div 2            # essayer au milieu
        selonQue                                       # plus petit / plus grand / égal ?
            t.tbEnt[milieu] > v :
                bDroite ← milieu - 1                  # trop grand, nouvelle droite
            t.tbEnt[milieu] < v :
                bGauche ← milieu + 1                  # trop petit, nouvelle gauche
            Sinon
                trouvé ← vrai                          # trouvé à l'indice milieu
        finSQ
        fini ← (bGauche > bDroite)                    # fin du processus ?
    finTQ
    Si trouvé Alors renvoie(milieu)                   # indice
        Sinon renvoie(0)
    finSi
Fin

Début
    chargerTableau(tb1)
    trierTableau(tb1)
    lire(valeur)                                       # que cherche-t-on ?
    indiceRetour ← rechercheDichotomique(tb1, valeur)  # chercher
    Si indiceRetour ≠ 0
        Alors écrire("trouvé ", valeur, " à l'indice ", indiceRetour)
        Sinon écrire("pas trouvé ", valeur)
    finSi
Fin.
```

e) TRIER OU INDEXER ?

On peut retenir deux raisons majeures pour lesquelles on souhaite conserver les éléments d'un tableau dans un ordre particulier et précis :

- la présentation des données (affichage, impression ...)
- l'accélération de la recherche (la recherche séquentielle s'arrête plus vite en cas d'échec et la recherche dichotomique est particulièrement plus efficace).

Il y a deux techniques fondamentales pour assurer un 'ordre' entre les éléments d'un tableau :

- le tri 'physique', consistant à modifier le tableau en imposant directement l'ordre entre les éléments eux-mêmes en les déplaçant si nécessaire ... cette technique vient d'être étudiée

```

Constante tSup = 100
Variable tabl : Tableau[tSup] de Texte
        bSup : 0..tSup
        ind : 1..tSup

```

...

tabl (avant)

1	2	3	4	5	6	7	8	9	10	11	12
jean	zoé	anne	anna	pierre	noël	henri	eva	justin			

tabl (après)

1	2	3	4	5	6	7	8	9	10	11	12
anna	anne	eva	henri	jean	justin	noël	pierre	zoe			

pour obtenir l'ordre (ou effectuer rapidement une recherche), il suffit d'itérer séquentiellement le tableau une fois trié ...

...

```
chargerTableau
```

```
trierTableau
```

```
Pour ind de 1 à bSup Faire : écrire(tabl[ind]) : finPr
```

...

- l'indexation, consistant à utiliser un tableau supplémentaire qui va conserver uniquement des indices du premier ; la technique consiste à placer dans ce second tableau les indices des éléments du premier dans lesquels ils doivent être 'lus' pour obtenir l'ordre souhaité

```

Variable tabl : Tableau[tSup] de Texte
        idx : Tableau[tSup] de 1..tSup
        ind : 1..tSup

```

...

tabl (avant indexation)

1	2	3	4	5	6	7	8	9	10	11	12
jean	zoé	anne	anna	pierre	noël	henri	eva	justin			

idx (avant indexation)

1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9			

tabl (après indexation : inchangé)

1	2	3	4	5	6	7	8	9	10	11	12
jean	zoé	anne	anna	pierre	noël	henri	eva	justin			

idx (après indexation)

1	2	3	4	5	6	7	8	9	10	11	12
4	3	8	7	1	9	6	5	2			

c'est en itérant séquentiellement le second tableau et en accédant à mesure à l'élément du premier tableau ainsi référencé que l'ordre entre les éléments réapparaît (en toute circonstance, le premier tableau reste inchangé)

...

```
chargerTableau
```

```
indexerTableau
```

```
Pour ind de 1 à bSup Faire : écrire(tabl[idx[ind]]) : finPr
```

comme on peut l'observer ci-dessous, les algorithmes de tri et d'indexation sont fort semblables (on a utilisé la technique de sélection consistant à rechercher à 'chaque tour' le plus petit élément et à l'échanger) ; on remarquera cependant côté indexation

- la nécessité d'initialiser correctement l'index (chaque élément de l'index désigne ainsi son correspondant dans le tableau principal)
- la comparaison entre éléments du tableau à travers l'index (`tab[idx[ind]]`)
- l'échange entre les éléments de l'index (indices du tableau principal)

Variable `tab` : Tableau[tSup] de Texte  
`idx` : Tableau[tSup] de Entier

#### Algo de tri

```
Procédure triSélection :
Variable ech : Entier
      ind1, ind2, indMin : Entier
Début

Pour ind1 de 1 à bSup - 1 Faire
  indMin ← ind1
  Pour ind2 de ind1 + 1 à bSup Faire
    Si tab[ind2] < tab[indMin] Alors
      indMin ← ind2
  finSi
finPr
Si indMin ≠ ind1 Alors
  ech ← tab[ind1]
  tab[ind1] ← tab[indMin]
  tab[indMin] ← ech
finSi
finPr
Fin
```

#### Algo d'indexation

```
Procédure indexSélection :
Variable ech : Entier
      ind1, ind2, indMin : Entier
Début
  Pour ind1 de 1 à bSup Faire # !! initialisation !!
    idx[ind1] ← ind1
  finPr
  Pour ind1 de 1 à bSup - 1 Faire
    indMin ← ind1
    Pour ind2 de ind1 + 1 à bSup Faire
      Si tab[idx[ind2]] < tab[idx[indMin]] Alors
        indMin ← ind2
    finSi
  finPr
  Si indMin ≠ ind1 Alors
    ech ← idx[ind1]
    idx[ind1] ← idx[indMin]
    idx[indMin] ← ech
  finSi
finPr
Fin
```

Tout milite en faveur de l'indexation :

- lorsque l'on trie physiquement un tableau de données, on doit échanger ses éléments; que ceux-ci soient de 'simples' chaînes de caractères ou plus complexes encore, des enregistrements (tableaux de personnes, p.ex.), ces échanges sont très coûteux en affectations (échanger deux chaînes ou deux enregistrements représente d'importants mouvements d'octets en mémoire)
- par contre l'indexation laisse le tableau de données inchangé et ce sont uniquement les indices (de simples entiers donc) qui sont échangés dans le tableau-index
- dans le cas de tableaux d'enregistrements, trier physiquement privilégie l'ordre pour un champ (ou une combinaison de champs) au détriment des autres, ce qui peut entraîner des (ré)exécutions de tri très coûteuses (p.ex. si un tableau de personnes doit être présenté aussi bien par ordre alphabétique des noms que par ordre croissant des codes postaux)
- l'indexation par contre peut faire cohabiter plusieurs index indépendants les uns des autres (autant que nécessaire) et toujours sans toucher au tableau de données lui-même
- enfin - s'il faut vraiment en trouver un – le reproche que l'on peut adresser à l'indexation est son (sur)coût en espace-mémoire : un tableau d'entiers supplémentaire est nécessaire par index

un exemple final récapitulatif ? multi-indexation d'un tableau de personnes : on souhaite pour-voir à tout moment afficher la liste de personnes dans l'ordre alphabétique des noms (et dans l'ordre croissant des âges pour les doublons) ou cette liste dans l'ordre croissant des âges (et dans l'ordre alphabétique des noms en cas d'égalité); on en profite pour un usage 'massif' de procédures et fonctions paramétrées

```

Constante maxPers = 50                                # taille physique
Type TPersonne = Enregistrement                       # enregistrement nom + âge
    nom : Texte                                       # composant nom
    age : Entier                                     # composant âge
finEnr
Type TtbPersonnes = Tableau[maxPers] de TPersonne    # tableau physique
Type TtbpPersonnes = Enregistrement                 # tableau partiel
    nbPers : Entier                                 # taille logique
    tbPers : TtbPersonnes                          # tableau
finEnr
Type TIndex = Tableau[maxPers] de Entier             # tableau index

```

```

Variable tbpPersonnes : TtbpPersonnes                # variable tableau de personnes
    idxNomsAges, idxAgesNoms : TIndex                # et ses deux index

```

tableau de données tbpPersonnes (après encodage)

1	2	3	4	5	6	7	8	9	10	11	...
jean	zoé	anne	anna	pierre	noël	henri	anne	jean			
31	17	31	71	67	60	41	17	25			

index idNomsAges (après indexation)

1	2	3	4	5	6	7	8	9	10	11	...
4	8	3	7	9	1	6	5	2			

index idxAgesNoms (après indexation)

1	2	3	4	5	6	7	8	9	10	11	...
8	2	9	3	1	7	6	5	4			

```

Procédure initPersonnes(ref tp : TtbpPersonnes) :

```

```

# initialise le tableau de personnes tp en mettant son compteur à 0

```

```

Début

```

```

    tp.nbPers ← 0

```

```

Fin

```

```

Fonction lirePersonne() : TPersonne

```

```

# saisie et renvoi d'un enregistrement de type personne

```

```

Variable p : TPersonne

```

```

Début

```

```

    lire(p.nom)      # à valider éventuellement

```

```

    lire(p.age)      # à valider éventuellement

```

```

    renvoie p        # 'valeur' renvoyée

```

```

Fin

```

```

Procédure ajoutPersonne(ref tp : TtbpPersonnes, ref fini : Logique) :

```

```

# ajout d'une personne dans le tableau tp : si c'est possible !

```

```

Variable p : TPersonne

```

```

Début

```

```

    p ← lirePersonne()    # généralisation de l'affectation

```

```

    fini ← tp.nbPers = maxPers OU p.nom = "*****"

```

```

    Si non fini Alors

```

```

        tp.nbPers ← tp.nbPers + 1 # ajouter une personne ...

```

```

        tp.tbPers[tp.nbPers] ← p # ... à la suite dans le tableau

```

```

    finSi

```

```

Fin

```

```

Procédure chargerPersonnes(ref tp : TtbpPersonnes) :

```

```

# remplir le tableau de personnes tp

```

```

Variable fini : Logique

```

```

Début

```

```

    Répéter

```

```

        ajoutPersonne(tp, fini)

```

```

    Jusqu'à fini

```

```

Fin

```

```

Procédure initIndex(ref idx : TIndex) :
# initialiser n'importe quel tableau-index idx
Variable indice : Entier
Début
  Pour indice de 1 à maxPers Faire
    idx[indice] ← indice
  finPr
Fin

Procédure échangerIndex(ref i1 : Entier, i2 : Entier) :
# échanger les valeurs de deux entiers (donc dans un index, p.ex.)
Variable tmp : Entier
Début
  tmp ← i1
  i1 ← i2
  i2 ← tmp
Fin

Procédure indexerAgeNom(tp : TtbpPersonnes, ref idx TIndex) :
# créer l'index idx sur le tableau tp (champs nom + âge)
Variable ind1, ind2, indMin : Entier
Début
  initIndex(idx)
  Pour ind1 de 1 à tp.nbPers - 1 Faire
    indMin ← ind1
    Pour ind2 de ind1 + 1 à tp.nbPers Faire
      Si tp[idx[ind2]].nom < tp[idx[indMin]].nom OU
        (tp[idx[ind2]].nom = tp[idx[indMin]].nom ET
          tp[idx[ind2]].age < tp[idx[indMin]].age) Alors
        indMin ← ind2
      finSi
    finPr
    Si indMin <> ind1 Alors échangerIndex(idx[indMin], idx[ind1]) finSi
  finPr
Fin

Procédure indexerAgeNom(tp : TtbpPersonnes, ref idx TIndex) :
# créer l'index idx sur le tableau tp (champs âge + nom)
Variable ind1, ind2, indMin : Entier
Début
  initIndex(idx)
  Pour ind1 de 1 à tp.nbPers - 1 Faire
    indMin ← ind1
    Pour ind2 de ind1 + 1 à tp.nbPers Faire
      Si tp[idx[ind2]].age < tp[idx[indMin]].age OU
        (tp[idx[ind2]].age = tp[idx[indMin]].age ET
          tp[idx[ind2]].nom < tp[idx[indMin]].nom) Alors
        indMin ← ind2
      finSi
    finPr
    Si indMin <> ind1 Alors échangerIndex(idx[indMin], idx[ind1]) finSi
  finPr
Fin

Procédure afficherPersonnes(tp : TtbpPersonnes, idx TIndex) :
# afficher le contenu du tableau tp à travers l'index idx
Variable ind : Entier
Début
  Pour ind de 1 à tp.nbPers Faire
    écrire(tp[idx[ind]].nom, tp[idx[ind]].age)
  finPr
Fin

```

*# algorithme : séquence principale*

Début

```

initPersonnes(tbpPersonnes)           # vider tableau de personnes
chargerPersonnes(tbpPersonnes)         # emplir tableau de personnes
indexerNom(tbpPersonnes, idxNomsAges)   # indexer sur noms + âges
indexerAgeNom(tbpPersonnes, idxAgesNoms) # indexer sur âges + noms
afficherPersonnes(tbpPersonnes, idxNomsAges) # afficher via index
afficherPersonnes(tbpPersonnes, idxAgesNoms) # idem

```

Fin.

## f) GÉNÉRICITÉ DU TRI, DE L'INDEXATION ET DE LA RECHERCHE

À partir du moment où l'on utilise des procédures et des fonctions paramétrées, avec la possibilité de surcharge, on peut envisager d'écrire une fois pour toutes une et une seule procédure pour pouvoir trier ou indexer n'importe quel (type de) tableau (c.-à-d. quelle que soit la structure de ses éléments), et de même de rédiger une et une seule fonction de recherche ...

Cette possibilité de s'adapter à n'importe quel contenu porte le nom technique de généricité

Mais pas si vite ... pour être générique, un algorithme doit être totalement indépendant de la structure de données ... et ce que nous avons écrit jusqu'ici comporte – malgré l'effort louable de paramétrisation – pas mal de dépendances :

- la première provient paradoxalement de la déclaration d'un tableau partiel (tableau physique + taille) sous forme d'enregistrement

```

Type TPersonne = Enregistrement
    nom : Texte
    age : Entier
finEnr
Type TtbPersonnes = Tableau[100] de TPersonne
Type TtbpPersonnes = Enregistrement
    nbPers : Entier
    tbPers : TtbPersonnes
finEnr
Variable tbP : TtbpPersonnes

Type TtbMesures = Tableau[100] de Réel
Type TtbpMesures = Enregistrement
    nbMes : Entier
    tbMes : TtbMesures
finEnr
Variable tbM : TtbpMesures

```

lors de la rédaction de la procédure de tri par exemple, on trouve deux itérateurs imbriqués dont les bornes dépendent du nom d'un champ du paramètre : la taille du tableau (nombre d'éléments) : il faut donc rédiger une procédure spécifique par type de tableau

```

Procédure tri(ref t : TtbpPersonnes) :
Variable ind1, ind2, indMin : Entier
Début
    Pour ind1 de 1 à t.nbPers - 1 Faire
        indMin ← ind1
        Pour ind2 de ind1 + 1 à t.nbPers Faire
            # ... etc ...
Fin

Procédure tri(ref t : TtbpMesures) :
Variable ind1, ind2, indMin : Entier
Début
    Pour ind1 de 1 à t.nbMes - 1 Faire
        indMin ← ind1
        Pour ind2 de ind1 + 1 à t.nbMes Faire
            # ... etc ...
Fin

```



l'invocation est cependant identique (grâce à la généricité) :

```
tri(tbP)
tri(tbM)
```

pour s'affranchir de cette dépendance, la solution consiste à passer deux paramètres distincts à la procédure : le tableau physique d'une part, sa taille (son nombre d'éléments) d'autre part

```
Procédure tri(ref t : TtbPersonnes, n : Entier) :
Variable ind1, ind2, indMin : Entier
Début
  Pour ind1 de 1 à t.n - 1 Faire
    indMin ← ind1
    Pour ind2 de ind1 + 1 à t.n Faire
# ... etc ...
Fin
```

```
Procédure tri(ref t : TtbMesures, n : Entier) :
Variable ind1, ind2, indMin : Entier
Début
  Pour ind1 de 1 à t.n - 1 Faire
    indMin ← ind1
    Pour ind2 de ind1 + 1 à t.n Faire
# ... etc ...
Fin
```

invocation :

```
tri(tbP.tbPers, tbP.nbPers)
tri(tbM.tbMes, tbM.nbMes)
```

- la seconde provient du 'cœur' de l'algorithme de tri : la comparaison non seulement est dépendante des noms des champs à comparer mais également du nombre de champs (critère majeur/mineur)

tri croissant sur valeur (tableau de mesures)

```
Si t[ind2] < t[indMin] Alors
  indMin ← ind2
finSi
```

tri croissant sur age + nom (tableau de personnes)

```
Si t[ind2].age < t[indMin].age OU
  (t[ind2].age = t[indMin].age ET
  t[ind2].nom < t[indMin].nom) Alors
  indMin ← ind2
finSi
```

tri croissant sur nom + age (tableau de mesures)

```
Si t[ind2].nom < t[indMin].nom OU
  (t[ind2].nom = t[indMin].nom ET
  t[ind2].age < t[indMin].age) Alors
  indMin ← ind2
finSi
```

la solution la plus élégante à cette problématique est de déléguer ce travail de comparaison à une fonction (qui portera le nom technique de comparateur), on lui passera comme paramètres deux éléments de tableaux à comparer (disons e1 et e2) et elle renverra une valeur entière selon le 'standard' suivant : -1 (e1 < e2), 0 (e1 = e2) ou +1 (e1 > e2)

il faut bien entendu écrire un comparateur spécifique pour chaque type d'élément de tableau (comme il faut écrire une procédure spécifique d'échange ...) ... mais ces fonctions et procédures pourront être réutilisées lors du tri, de l'indexation et de la recherche

## comparateur de mesures (en fait de Réels)

```

fonction compare(e1 : Réel, e2 : Réel) : Entier
Début
  selonQue
    e1 < e2 : renvoie -1
    e1 = e2 : renvoie 0
    Sinon   : renvoie +1
  finSQ
Fin

```

```

Procédure échanger(ref e1 : Réel, ref e2 : Réel) :
Variable tmp : Réel
Début
  tmp ← e1; e1 ← e2; e2 ← tmp
Fin

```

## comparateur de personnes (pour nom + age)

```

fonction compare(e1 : TPersonne, e2 : TPersonne) : Entier
Début
  selonQue
    e1.nom < e2.nom : renvoie -1
    e1.nom = e2.nom :
      selonQue
        e1.age < e2.age : renvoie -1
        e1.age = e2.age : renvoie 0
        Sinon           : renvoie +1
      finSQ
    Sinon : renvoie +1
  finSQ
Fin

```

```

Procédure échanger(ref e1 : TPersonne, ref e2 : TPersonne) :
Variable tmp : TPersonne
Début
  tmp ← e1; e1 ← e2; e2 ← tmp
Fin

```

A présent, un dernier coup d'œil sur le tri de mesures ou de personnes :

```

Procédure tri(ref t : TtbMesures, n : Entier) :
Variable ind1, ind2, indMin : Entier
Début
  Pour ind1 de 1 à t.n - 1 Faire
    indMin ← ind1
    Pour ind2 de ind1 + 1 à t.n Faire
      Si compare(t[ind2], t[indMin]) Alors indMin ← ind2 finSi
    finPr
    Si indMin ≠ ind1 Alors échanger(t[indMin], t[ind1]) finSi
  finPr
Fin

```

```

Procédure tri(ref t : TtbPersonnes, n : Entier) :
Variable ind1, ind2, indMin : Entier
Début
  Pour ind1 de 1 à t.n - 1 Faire
    indMin ← ind1
    Pour ind2 de ind1 + 1 à t.n Faire
      Si compare(t[ind2], t[indMin]) Alors indMin ← ind2 finSi
    finPr
    Si indMin ≠ ind1 Alors échanger(t[indMin], t[ind1]) finSi
  finPr
Fin

```

## g) NOTION DE FUSION

Basiquement, la fusion de deux tableaux consiste à en constituer un troisième à partir de leur(s) contenu(s) : il n'y a a priori rien de très difficile à cela (il faut tout de même vérifier que le tableau-cible possède la place nécessaire ...

Avec le formalisme des tableaux partiels (de personnes, p.ex.), la procédure de fusion est simplissime à écrire :

```

Constante maxPers = 50                                # taille physique
Type TPersonne = Enregistrement                       # enregistrement nom + âge
    nom : Texte                                         # composant nom
    age : Entier                                        # composant âge
finEnr
Type TtbPersonnes = Tableau[maxPers] de TPersonne     # tableau physique
Type TtbpPersonnes = Enregistrement                  # tableau partiel
    nbPers : Entier                                    # taille logique
    tbPers : TtbPersonnes                             # tableau
finEnr

```

Variable, tbP1, tbP2, tbP3 : TtbpPersonnes

tableau de données tbP1

1	2	3	4	5	6	7	8	9	10	11	...
jean	zoé	anne	anna	pierre	noël						
31	17	31	71	67	60						

tableau de données tbP2

1	2	3	4	5	6	7	8	9	10	11	...
henri	anne	jean									
41	17	25									

tableau de données tbP3 (après fusion)

1	2	3	4	5	6	7	8	9	10	11	...
jean	zoé	anne	anna	pierre	noël	henri	anne	jean			
31	17	31	71	67	60	41	17	25			

Procédure fusionnerTableaux(t1, t2 : TtbpPersonnes, ref t3 : TtbpPersonnes) :

Variable indice : Entier

Début

Si t1.nbPers + t2.nbPers ≤ maxPers Alors # assez de place ?

Pour indice de 1 à t1.nbPers # copier t1 au début de t3

t3[indice] ← t1[indice]

finPr

Pour indice de 1 à t2.nbPers # compléter ensuite avec t2

t3[t1.nbPers + indice] ← t2[indice] # regardez l'indexage de t3

finPr

t3.nbPers ← t1.nbPers + t2.nbPers # ne pas oublier : taille de t3

finSi

Fin

Si on en restait là, il ne serait nécessaire de 'déranger' ce nouveau concept ni ce sous-chapitre ... le plus souvent, la fusion consiste à combiner deux tableaux triés en un troisième lui aussi trié : on évitera évidemment de procéder comme précédemment et de trier le tableau résultant (le coût serait prohibitif<sup>70</sup>)

tableau de données trié tbP1

1	2	3	4	5	6	7	8	9	10	11	...
anna	anne	jean	pierre	noël	zoé						
71	31	31	67	60	17						

tableau de données trié tbP2

1	2	3	4	5	6	7	8	9	10	11	...
anne	henri	jean									
17	41	25									

Comment procéder alors ?

<sup>70</sup> si tb1 et tb2 ont chacun N éléments, les trier coûte N<sup>2</sup> pour chacun, à quoi on ajouterait encore (2N)<sup>2</sup> pour le tri de tb3 !!!

On va utiliser un itérateur distinct pour chaque tableau : tous deux démarrent évidemment à l'indice 1; puisque les deux tableaux tbP1 et tbP2 sont triés (on suppose en ordre croissant), à cet indice figure l'élément le plus petit de chaque tableau : il suffit donc de les comparer, de placer dans tbP3 (lui aussi a un itérateur qui démarre à 1 et est incrémenté au fur et à mesure) le plus petit des deux et de faire 'avancer' l'itérateur correspondant ... on continue ainsi jusqu'à épuisement d'un des deux tableaux; à ce moment, il n'y a qu'à recopier le 'solde' du tableau restant

### étape 1

tableau de données trié tbP1

1	2	3	4	5	6	7	8	9	10	11	...
anna	anne	jean	noël	pierre	zoé						
71	31	31	60	67	17						

tableau de données trié tbP2

1	2	3	4	5	6	7	8	9	10	11	...
anne	henri	jean									
17	41	25									

tableau de données tbP3

1	2	3	4	5	6	7	8	9	10	11	...
anna											
71											

### étape 2

tableau de données trié tbP1

1	2	3	4	5	6	7	8	9	10	11	...
anna	anne	jean	noël	pierre	zoé						
71	31	31	60	67	17						

tableau de données trié tbP2

1	2	3	4	5	6	7	8	9	10	11	...
anne	henri	jean									
17	41	25									

tableau de données tbP3

1	2	3	4	5	6	7	8	9	10	11	...
anna	anne										
71	17										

etc...

vient un moment où un tableau a été entièrement copié (ici tbP2)

tableau de données trié tbP1

1	2	3	4	5	6	7	8	9	10	11	...
anna	anne	jean	noël	pierre	zoé						
71	31	31	60	67	17						

tableau de données trié tbP2

1	2	3	4	5	6	7	8	9	10	11	...
anne	henri	jean									
17	41	25									

tableau de données tbP3

1	2	3	4	5	6	7	8	9	10	11	...
anna	anne	anne	henri	jean							
71	17	31	41	25							

on peut alors recopier le reste de l'autre tableau (ici tbP1)

tableau de données trié tbP1

1	2	3	4	5	6	7	8	9	10	11	...
anna	anne	jean	noël	pierre	zoé						
71	31	31	60	67	17						

tableau de données trié tbP2

1	2	3	4	5	6	7	8	9	10	11	...
anne	henri	jean									
17	41	25									

tableau de données tbP3

1	2	3	4	5	6	7	8	9	10	11	...
anna	anne	anne	henri	jean	jean	noël	pierre	zoé			
71	17	31	41	25	31	60	67	17			

Il y a plusieurs écritures possibles de cet algorithme, mais la plus classique est par exemple

```

Procédure fusionnerTableaux(t1, t2 : TtbpPersonnes, ref t3 : TtbpPersonnes) :
Variable indice, ind1, ind2 : Entier
      fini1, fini2 : Logique
Début
  t3.nbPers ← t1.nbPers + t2.nbPers           # pour ne pas oublier !
  indice ← 0                                  # pour gérer l'ajout dans t3
  ind1 ← 1
  fini1 ← (ind1 > t1.nbPers)                  # prévoir tableau vide !
  ind2 ← 1
  fini2 ← (ind2 > t2.nbPers)                  # prévoir tableau vide !
  tantQue non (fini1 ou fini2)                # arrêt sur l'un ou l'autre
    indice ← indice + 1                       # un élément de plus dans t3
    Si t1[ind1].nom < t2[ind1].nom OU         # lequel ?
      (t1[ind1].nom = t2[ind1].nom ET t1[ind1].age ≤ t2[ind1].age) Alors
      t3[indice] ← t1[ind1]                   # celui de t1
      ind1 ← ind1 + 1                         # avancer dans t1
      fini1 ← (ind1 > t1.nbPers)              # fini t1 ?
    Sinon
      t3[indice] ← t2[ind2]                   # celui de t2
      ind2 ← ind2 + 1                         # avancer dans t2
      fini2 ← (ind2 > t1.nbPers)              # fini t2 ?
    finSi
  finTQ
  Si fini2 Alors
    Pour ind1 ← ind1 à t1.nbPers Faire        # solder le reste du premier
      indice ← indice + 1
      t3[indice] ← t1[ind1]
    finPr
  Sinon
    Pour ind2 ← ind2 à t2.nbPers Faire        # solder le reste du deuxième
      indice ← indice + 1
      t3[indice] ← t2[ind2]
    finPr
  finSi
Fin

```



3. Lors d'élections, chaque candidat obtient par tirage au sort un numéro de liste (1, 2, 3 ... qu'on utilise comme indice de tableau) ;

Côté données, on dispose de deux tableaux distincts, l'un conservant les noms des candidats, l'autre comptabilisant les votes obtenus pour chacun

Constante nbCand = 5

Variable candidats : Tableau[nbCand] de Texte

votes : Tableau[nbCand] de Entier

*Exemple*

Tableau 1 : les candidats

Céline	Fabre	Jeanne	Louis	Bernard		
--------	-------	--------	-------	---------	--	--

Tableau 2 : les votes obtenus

98	120	185	67	103		
----	-----	-----	----	-----	--	--

Écrire les procédures permettant (avec toutes les validations nécessaires)

- d'exprimer le vote pour le candidat de son choix
- quand le vote est terminé, de trier les tableaux de manière à présenter résultats et candidats dans l'ordre décroissant du nombre de votes obtenus





## 5.6. POUR CONCLURE (?) : EXEMPLES

### a) INTRODUCTION

Examinons le cas suivant :

Un magasin d'informatique souhaite une (mini) gestion du stock des produits qu'il propose à la vente (et on se limitera pour faire simple aux disques durs, clés usb, media CD et DVD);

Ce stock varie constamment suite aux achats (entrées) et aux ventes (sorties);

On désire simplement gérer ces entrées/sorties et bien entendu consulter l'état du stock pour un article particulier et afficher/imprimer un catalogue

Une petite conversation avec le gérant conduit à la définition d'un article : il possède les caractéristiques suivantes :

- un libellé (chaîne de caractères) :
- une capacité (en Gb, ex 500) ou une vitesse maximum de lecture (en x, ex. 52) (un entier)
- un prix (type réel)
- et bien entendu une quantité (entière) en stock

On peut donc imaginer dans un premier temps une structure multi-tableaux :

#### variante 1

```

Constante maxArt = 100                                # nombre max d'articles
Variable tablLibellés : Tableau[maxArt] de Texte       # libellés
      tabCapacités : Tableau[maxArt] de Entier         # capacités/vitesses
      tabPrix      : Tableau[maxArt] de Réel          # prix
      tabQuantités : Tableau[maxArt] de Entier         # quantité en stock
      nbArt : Entier                                   # borne : nombre articles
  
```

mais rien n'indique clairement qu'il s'agit là d'un ensemble cohérent et qu'un article est défini par l'ensemble des cases des quatre tableaux qui partagent le même indice; toutes les opérations devront être multi-tableaux

On privilégiera donc une approche par enregistrement (le type d'enregistrement définit un modèle abstrait d'article avec ses quatre champs et on a donc un seul tableau d'enregistrements)

#### variante 2

```

Constante maxArt = 100                                # nombre max d'articles
Type TArticle = Enregistrement                        # article abstrait
  libellé : Texte
  capacité : Entier
  prix : Réel
  quantité : Entier
finEnr
Variable tablStock : Tableau[maxArt] de TArticle      # tableau d'articles
      nbArt : Entier                                   # borne : nombre articles
  
```

Rédigeons à présent quelques procédures (et fonctions !) pour ces deux implémentations :

- pour consulter ou pour faire entrer ou pour faire sortir un article de capacité/vitesse donnée, il faut d'abord le trouver !
- dans le cas de l'ajout si on le trouve, on augmente sa quantité en stock et si on ne le trouve pas, c'est un nouvel article à créer
- dans le cas de la suppression, si on le trouve on diminue sa quantité en stock sinon on signale l'insuccès)
- dans le cas de la consultation, si on le trouve on obtient sa quantité en stock sinon on signale l'insuccès)
- on commence donc par écrire une fonction de recherche; pourquoi une fonction ? parce qu'elle va retourner la valeur de l'indice en cas de succès et disons 0 en cas d'échec; on lui passera comme paramètres d'entrée le libellé et la capacité/vitesse de l'article cherché

variante 1 (multitablesaux)

```

Fonction chercheArticle(lib : Texte, capa : Entier) : typIndArt
Variable indice : 1..maxArt+1
      trouvé, fini : Logique
Début
  trouvé ← faux
  indice ← 1
  fini ← (indice > nbArt)
  tantQue non (trouvé ou fini)
    Si tablLibellés[indice] = lib ET tablCapacités[indice] = capa Alors # 2 critères
      trouvé ← vrai
    Sinon
      indice ← indice + 1
      fini ← (indice > nbArt)
    finSi
  Si trouvé Alors renvoie(indice)
    Sinon renvoie(-1)
  finSi
Fin

```

variante 2 (tableau d'enregistrements)

```

Fonction chercheArticle(lib : Texte, capa : Entier) : typIndArt
Variable indice : 1..maxArt+1
      trouvé, fini : Logique
Début
  trouvé ← faux
  indice ← 1
  fini ← (indice > nbArt)
  tantQue non (trouvé ou fini)
    Avec tablStock[indice]
      Si libellé = lib ET capacité = capa Alors # 2 critères
        trouvé ← vrai
      Sinon
        indice ← indice + 1
        fini ← (indice > nbArt)
      finSi
    finAvec
  Si trouvé Alors renvoie(indice)
    Sinon renvoie(-1)
  finSi
Fin

```

A présent, la procédure d'ajout

variante 1 (multitablesaux)

```

Procédure ajouterArticle :
Variable libellé : Texte, capacité : Entier, prix : Réel, quantité : Entier
      indice : typIndArt
Début
  lire(libellé)
  lire(capacité)
  indice ← chercheArticle(libellé, capacité) # où est cet article ?
  Si indice = 0 Alors # pas trouvé ! le créer alors ...
    lire(prix)
    lire(quantité)
    nbArt ← nbArt + 1 # un article en plus dans le stock
    tablLibellés[nbArt] ← libellé # le mettre derrière les autres
    tablCapacités[nbArt] ← capacité
    tablPrix[nbArt] ← prix
    tablQuantités[nbArt] ← quantité
  Sinon # trouvé ! augmenter le stock
    lire(quantité)
    tablQuantités[indice] ← tablQuantités[indice] + quantité
  finSi
Fin

```

variante 2 (tableau d'enregistrements)

```

Procédure ajouterArticle :
Variable art : tArt           # un article de travail
    indice : typIndArt
Début
    lire(art.libellé)
    lire(art.capacité)
    indice ← chercheArt(art.libellé, art.capacité) # où est cet article ?
Si indice = 0 Alors           # pas trouvé ! le créer alors ...
    lire(art.prix)
    lire(art.quantité)
    nbArt ← nbArt + 1          # un article en plus dans le stock
    tablStock[nbArt] ← art     # le mettre derrière les autres
Sinon                         # trouvé ! augmenter le stock
    lire(art.quantité)
    tablStock[indice].quantité ← tablStock[indice].quantité + art.quantité
finSi
Fin

```

Continuons avec une procédure de tri composite (en majeur sur le libellé et en mineur sur la capacité/vitesse)

variante 1 (multitables)

```

Procédure triCatalogue :
Variable ind1, ind2, indMin : typIndArt
    ech : Texte, echE : Entier, echR : Réel           # variables pour échange
Début
    Pour ind1 de 1 à nbArt - 1
        indMin ← ind1
        Pour ind2 de ind1 + 1 à nbArt
            Si tablLibellés[ind2] < tablLibellés[indMin] OU # majeur/mineur en une fois
                (tablLibellés[ind2] = tablLibellés[indMin]
                    ET tablCapacités[ind2] < tablCapacités[indMin]) Alors
                    indMin ← ind2
            finSi
        finPour
        Si indMin ≠ ind1 Alors # échanger dans quatre tableaux !!!!
            echT ← tablLibellés[indMin]
            tablLibellés[indMin] ← tablLibellés[ind1]
            tablLibellés[ind1] ← echT
            echE ← tablCapacités[indMin]
            tablCapacités[indMin] ← tablCapacités[ind1]
            tablCapacités[ind1] ← echE
            echR ← tablPrix[indMin]
            tablPrix[indMin] ← tablPrix[ind1]
            tablPrix[ind1] ← echR
            echE ← tablQuantités[indMin]
            tablQuantités[indMin] ← tablQuantités[ind1]
            tablQuantités[ind1] ← echE
        finSi
    finPour
Fin

```

variante 2 (tableau d'enregistrements)**Procédure** triCatalogue :

Variable ind1, ind2, indMin : typIndArt

**echA** : tArticle

# variable article pour échange

Début

Pour ind1 de 1 à nbArt - 1

indMin ← ind1

Pour ind2 de ind1 + 1 à nbArt

Si tablStock[ind2].libellé < tablStock[indMin].**OU** # majeur/mineur en une fois

(tablStock[ind2].libellé = tablStock[indMin].libellé

**ET** tablStock[ind2].capacité < tablStock[indMin].capacité) Alors

indMin ← ind2

finSi

finPour

Si indMin ≠ ind1 Alors

# échanger deux articles du tableau**echA** ← tablStock[indMin]

tablStock[indMin] ← tablStock[ind1]

tablStock[ind1] ← **echA**

finSi

finPour

Fin

b) EXEMPLE(S) COMPLET(S)

On peut donc à présent imaginer l'algorithme complet de la variante 2 (tableau d'enregistrements puisqu'une implémentation multitableaux présente trop d'inconvénients)

```

Algorithme gestionStock1 :
Constante maxArt = 100                                # nombre max d'articles
Type tArticle = enregistrement                         # article abstrait
    libellé : Texte
    capacité : Entier
    prix : Réel
    quantité : Entier
Fin
Type typIndArt = 0..maxArt                             # type d'indice
Variable tablStock : tableau[maxArt] de tArticle # tableau d'articles
    nbArt : typIndArt                                # borne : nombre d'articles
    choix : Entier                                    # choix de menu
    terminé : Logique                                # état

Fonction chercheArticle(lib : Texte, capa : Entier) : typIndArt
Variable indice : 1..maxArt+1
    trouvé, fini : Logique
Début
    trouvé ← faux
    indice ← 1
    fini ← (indice > nbArt)
    tantQue non (trouvé ou fini)
        Avec tablStock[indice]
            Si libellé = lib ET capacité = capa Alors      # 2 critères
                trouvé ← vrai
            Sinon
                indice ← indice + 1
                fini ← (indice > nbArt)
        finSi
    finAvec
    Si trouvé Alors renvoie(indice) Sinon renvoie(-1) finSi
Fin

Procédure ajouterArticle :
Variable art : tArt                                    # un article de travail
    ind : typIndArt
Début
    lire(art.libellé)                                    # que cherche-t-on ?
    lire(art.capacité)
    ind ← chercheArticle(art.libellé, art.capacité) # où est cet article ?
    Si ind = 0 Alors                                     # pas trouvé ! le créer alors ...
        lire(art.prix)
        lire(art.quantité)
        nbArt ← nbArt + 1                                # un article en plus dans le stock
        tablStock[nbArt] ← art                            # le mettre derrière les autres
    Sinon                                                 # trouvé ! augmenter le stock
        lire(art.quantité)
        tablStock[ind].quantité ← tablStock[ind].quantité + art.quantité
    finSi
Fin

Procédure supprimerArticle :
Variable art : tArt                                    # un article de travail
    ind : typIndArt
Début
    lire(art.libellé)                                    # que cherche-t-on ?
    lire(art.capacité)
    ind ← chercheArticle(art.libellé, art.capacité) # où est cet article ?
    Si ind = 0 Alors                                     # pas trouvé ! le dire
        écrire('article inexistant !')
    Sinon                                                 # trouvé ! diminuer le stock ...
        lire(art.quantité)
        Si tablStock[ind].quantité < art.quantité Alors # ... si c'est possible !
            écrire('stock insuffisant !')
        Sinon
            tablStock[ind].quantité ← tablStock[ind].quantité - art.quantité
        finSi
    finSi
Fin

```

**Procédure consulterArticle :**

```

Variable art : tArt                # un article de travail
      ind : typIndArt
Début
  lire(art.libellé)                # que cherche-t-on ?
  lire(art.capacité)
  ind ← chercheArticle(art.libellé, art.capacité) # où est cet article ?
  Si ind = 0 Alors                 # pas trouvé ! le dire
    écrire('article inexistant !')
  Sinon                            # trouvé ! afficher détails ...
    écrire(tablStock[ind].prix)
    écrire(tablStock[ind].quantité)
  finSi
Fin

```

**Procédure trierCatalogue :**

```

Variable ind1, ind2, indMin : typIndArt
      echA : tArticle              # variable article pour échange
Début
  Pour ind1 de 1 à nbArt - 1
    indMin ← ind1
    Pour ind2 de ind1 + 1 à nbArt
      Si tablStock[ind2].libellé < tablStock[indMin].OU      # majeur ...
        (tablStock[ind2].libellé = tablStock[indMin].libellé # mineur
        et tablStock[ind2].capacité < tablStock[indMin].capacité) Alors
          indMin ← ind2
    finSi
  finPour
  Si indMin ≠ ind1 Alors                                     # échanger deux articles du tableau
    echA ← tablStock[indMin]
    tablStock[indMin] ← tablStock[ind1]
    tablStock[ind1] ← echA
  finSi
finPour
Fin

```

**Procédure afficherCatalogue :**

```

Variable ind : typIndArt
Début
  Pour ind de 1 à nbArt
    Avec tablStock[ind]
      écrire(catégorie)    # préfixage ...
      écrire(libellé)      # ... des champs
      écrire(prix)
      écrire(quantité)
    finAvec
  finPour
Fin

```

**Procédure afficherMenu :**

```

Début
  écrire('Gestion de stock')
  écrire('=====')
  écrire('0. Quitter')
  écrire('1. Créer')
  écrire('2. Ajout')
  écrire('3. Suppression')
  écrire('4. Consultation')
  écrire('5. Impression catalogue')
  écrire('>> votre choix >>')
  Répéter
    lire(choix)
  Jusque choix entre 0 et 5    # il vaudrait mieux mettre des constantes
Fin;

```

**Procédure traiterChoixMenu :**

```

Début
  selonQue choix Vaut
    0 : terminé ← vrai
    1 : créerStock
    2 : ajouterArticle
    3 : supprimerArticle
    4 : consulterArticle
    5 : trierCatalogue
      afficherCatalogue
  finSQ
Fin;

```

et enfin, la séquence principale, toujours minimaliste :

```
Début
  Répéter
    afficherMenu
    traiterChoixMenu
  Jusque terminé
Fin.
```

Peut-on faire mieux encore ? bien entendu ! comme on l'aura remarqué,

- la structure de données est plus élégante avec les enregistrements, mais le stock nécessite deux variables : un tableau d'articles `tblStock` et la variable `nbArt` indiquant le nombre d'articles et le plus grand indice atteint dans ce tableau : on va donc devoir passer systématiquement ces deux variables à toutes les fonctions et procédures

une fois encore, la structure d'enregistrement va s'avérer utile en rassemblant ces deux facettes d'un stock (tableau d'articles et nombre d'articles) en une seule structure :

```
Constante maxArt = 100                                # nombre max d'articles
Type tArticle = enregistrement                         # article abstrait
  libellé : Texte
  capacité : Entier
  prix : Réel
  quantité : Entier
Fin
Type typIndArt = 0..maxArt                             # type d'indice
Type tStock = Enregistrement                          # stock abstrait, composé
  tbStock : tableau[maxArt] de tArticle                # d'un tableau d'articles
  nbArt : typIndArt                                    # d'un nombre d'articles
Fin
Variable stock : tStock                                # stock concret
  erreur : 0..999                                       # code d'erreur
```

- chacune des procédures écrites dans l'exemple manipule directement et explicitement les variables globales. Pour faire de tous ces outils des composants réutilisables, il suffit de les paramétrer : on veillera bien entendu à passer correctement ces adresses (par copie/valeur ou par référence/adresse) dans chaque cas de figure
- dans la foulée, et toujours dans un souci d'une meilleure indépendance de ces outils procéduraux, on va en retirer les instructions d'entrées-sorties (donc on va redécouper les procédures à l'aide d'autres fonctions et/ou procédures)

Algorithme gestionStock2 :

```
Constante maxArt = 100                                # nombre max d'articles
Type tArticle = enregistrement                         # article abstrait
  libellé : Texte
  capacité : Entier
  prix : Réel
  quantité : Entier
Fin
Type typIndArt = 0..maxArt                             # type d'indice
Type tStock = Enregistrement                          # stock abstrait, composé
  tbStock : tableau[maxArt] de tArticle                # d'un tableau d'articles
  nbArt : typIndArt                                    # d'un nombre d'articles
Fin
Variable stock : tStock                                # stock concret
  choix : Entier                                       # choix de menu
  terminé : Logique                                   # état
```

```
Procédure initialiserStock(ref s : tStock)
# initialiser un stock s
Début
  s.nbArt ← 0    # aucun article en stock
Fin
```

**Procédure gérerErreurs :***# centralisation de la gestion d'erreurs et des messages associés*

Début

selonQue erreur Vaut

1 : écrire('

2 : écrire('article inexistant !')

3 : écrire('stock insuffisant')

finSQ

Fin

**Fonction chercheArticle(a : tArticle, s : tStock) : typIndArt***# chercher un article a dans un stock s : renvoyer son indice ou 0*

Variable indice : 1..maxArt + 1

trouvé, fini : Logique

Début

trouvé ← faux

indice ← 1

fini ← (indice &gt; nbArt)

tantQue non (trouvé ou fini)

Avec s.tablStock[indice]

Si libellé = a.libellé ET capacité = a.capacité Alors

# 2 critères

trouvé ← vrai

Sinon

indice ← indice + 1

fini ← (indice &gt; s.nbArt)

finSi

finAvec

Si trouvé Alors renvoie(indice) Sinon renvoie(-1) finSi

Fin

**Fonction lireArticle() : tArticle***# lit des champs d'un article et renvoie cet article*

Variable art : tArticle # article travail

Début

Avec art

lire(catégorie)

lire(libellé)

lire(prix)

lire(quantité)

finAvec

renvoie(art)

Fin

**Procédure ajoutArticle(a : tArticle, ref s : tStock) :***# ajouter un article a dans un stock s*

Variable ind : typIndArt

Début

Avec s

Si nbArt = tMax Alors

erreur ← 1

Sinon

ind ← chercheArticle(a, s) # où est cet article ?

Si ind = 0 Alors

# pas trouvé ! le créer alors ...

nbArt ← nbArt + 1

# un article en plus dans le stock

tablStock[nbArt] ← a

# le mettre derrière les autres

Sinon

# trouvé ! augmenter le stock

tablStock[ind].quantité ← tablStock[ind].quantité + a.quantité

finSi

finAvec

finSi

Fin

**Procédure ajouterArticle(ref s : tStock) :***# ajouter un article dans un stock s*

Variable art : tArticle

Début

art ← lireArticle()

ajoutArticle(art, s)

gérerErreurs

Fin



```

Procédure suppressionArticle(a : tArticle, ref s : tStock) :
# supprimer un article a d'un stock s
Variable ind : typIndArt
Début
  ind ← chercheArticle(a, s) # où est cet article ?
  Si ind = 0 Alors # pas trouvé ! le signaler
    erreur ← 2
  Sinon # trouvé ! diminuer le stock ...
    Avec s.tablStock[ind]
      Si quantité < a.quantité Alors # ... si c'est possible !
        erreur ← 3 # non ? le signaler
      Sinon
        quantité ← quantité - a.quantité
      finSi
    finAvec
  finSi
Fin

Procédure ajouterArticle(ref s : tStock) :
# supprimer un article d'un stock s
Variable art : tArticle
Début
  art ← lireArticle()
  suppressionArticle(art, s)
  gérerErreurs
Fin

Procédure consultationArticle(a : tArticle, s : tStock) :
# obtenir les détails sur un article a d'un stock s
Variable ind : typIndArt
Début
  ind ← chercheArticle(a, s) # où est cet article ?
  Si ind = 0 Alors # pas trouvé ! le dire
    erreur ← 2
  Sinon # trouvé ! afficher détails ...
    Avec s.tablStock[ind]
      écrire(catégorie)
      écrire(libellé)
      écrire(prix)
      écrire(quantité)
    finAvec
  finSi
Fin

Procédure consulterArticle(s : tStock) :
# consulter un article d'un stock s
Variable art : tArticle
début
  art ← lireArticle()
  consultationArticle(art, s)
  gérerErreurs
Fin;

Procédure trierCatalogue(ref s : tStock) :
Variable ind1, ind2, indMin : typIndArt
  echA : tArticle # variable article pour échange
Début
  Avec s
    Pour ind1 de 1 à nbArt - 1
      indMin ← ind1
      Pour ind2 de ind1 + 1 à nbArt
        Si tablStock[ind2].libellé < tablStock[indMin].OU # majeur ...
          (tablStock[ind2].libellé = tablStock[indMin].libellé # mineur
            et tablStock[ind2].capacité < tablStock[indMin].capacité) Alors
            indMin ← ind2
        finSi
      finPour
      Si indMin ≠ ind1 Alors # échanger deux articles du tableau
        echA ← tablStock[indMin]
        tablStock[indMin] ← tablStock[ind1]
        tablStock[ind1] ← echA
      finSi
    finPour
  finAvec
Fin

```

```

Procédure afficherCatalogue(s : tStock) :
# afficher le catalogue d'un stock s
Variable ind : typIndArt
Début
  Avec s
    Pour ind de 1 à nbArt
      Avec tablStock[ind]      # préfixage ...
        écrire(catégorie)      # ... des champs
        écrire(libellé)
        écrire(prix)
        écrire(quantité)
      finAvec
    finPour
  finAvec
Fin;

Procédure afficherMenu :
Début
  écrire('Gestion de stock')
  écrire('=====')
  écrire('0. Quitter')
  écrire('1. Créer')
  écrire('2. Ajout')
  écrire('3. Suppression')
  écrire('4. Consultation')
  écrire('5. Impression catalogue')
  écrire('>> votre choix >>')
  Répéter
    lire(choix)
  Jusqu'à choix entre 0 et 5
Fin;

Procédure traiterChoixMenu :
Début
  selonQue choix Vaut      # enfin, seule référence à la variable globale
    0 : terminé ← vrai
    1 : créerStock(stock)
    2 : ajouterArticle(stock)
    3 : supprimerArticle(stock)
    4 : consulterArticle(stock)
    5 : trierCatalogue(stock)
      afficherCatalogue(stock)
  finSQ
Fin;

```

et enfin, la séquence principale, toujours minimaliste :

```

Début
  Répéter
    afficherMenu
    traiterChoixMenu
  Jusqu'à terminé
Fin.

```



- *estVide*(*t* : Texte) : Logique :

cette fonction renvoie le caractère vide ou non de la chaîne passée en argument

exemple d'utilisation :      si estVide(t) alors ... sinon ... finSi

- *estContenue*(*t1*, *t2* : Texte) : Logique :

cette fonction renvoie si oui/non la seconde chaîne est contenue dans la première

```
exemple d'utilisation :   txt1 ← 'interro d''algorithmique'
                          txt2 ← 'algo'
                          si estContenue(t1, t2) alors ... sinon ... finSi
```

- *position*(*t1*, *t2* : Texte) : 0..255 :

cette fonction renvoie la position de la chaîne 2 dans la chaîne 1 (la position du premier caractère en cas de réussite, la valeur 0 en cas d'échec)

[illegible]

- *concatenation*(*t1*, *t2* : Texte) : Texte :

cette fonction renvoie une chaîne qui est la mise bout à bout des deux chaînes transmises comme paramètres

```
exemple d'utilisation :  t1 ← 'algo'
                        t2 ← 'rithmique'
                        ecrire(concatenation(t1, t2)) # affiche : algorithmique
```

- *concatener*(*t1*, *t2* : Texte) :

cette procédure modifie la première chaîne en y concaténant la seconde

```
exemple d'utilisation :   t1 ← `algo`  
                           t2 ← `rithmique`  
ecrire(t1)                # affiche : algo  
ecrire(t2)                # affiche : rithmique  
concatener(t1, t2))      # affiche : algorithmique  
ecrire(t2)               # affiche : rithmique
```

- *extraction(t : texte ; début : 1..255, longueur : 1..255) : Texte :*

cette fonction renvoie la sous-chaine de caractères à partir de la position début et sur la longueur spécifiée

conditions et variantes :

- si la longueur spécifiée est 0, ce sont tous les caractères depuis la position début qui sont renvoyés
- si la position début est supérieure à la longueur de la chaîne c, une chaîne vide est renvoyée
- si la longueur spécifiée 'dépasse' le contenu de c, c'est comme si on avait spécifié une longueur 0

```
exemple d'utilisation : t ← 'interro de maths'
ecrire(extraction(t,12, 4)) # affiche : math
ecrire(extraction(t,22, 4)) # affiche :
ecrire(extraction(t,9, 44)) # affiche : de maths
```

- *remplacerCar(t : Texte, c : Caractère, pos : 1..255) :*

cette procédure remplace le caractère de `t` à la position `pos` par le caractère `c`

```
exemple d'utilisation : t ← 'interro de maths'
                        remplacerCar(t,'*', 5))
                        ecrire(txt)                # affiche : inte*ro de maths
```

- *remplacerTCar(t : Texte, c1, c2 : Caractère) :*

cette procédure remplace dans t toutes les occurrences du caractère c1 par le caractère c2

```
exemple d'utilisation : t ← 'interro d''algorithmique'
                        remplacerTCar(t,'r', 'R'))
                        ecrire(t) # affiche : inteRRo d'algoRithmique
```

- *remplacerOcc(t1, t2, t3 : Texte) :*

cette procédure remplace dans  $t_1$  la première occurrence de la sous-chaîne  $t_2$  par la chaîne  $t_3$

```
exemple d'utilisation :  txt ← 'interro d''algo'
                        remplacerOcc(txt, 'd''algo', 'de math'))
                        ecrire(txt)                                # affiche : interro de math
```

- *remplacerTOcc*(*t1*, *t2*, *t3* : Texte) :

cette procédure remplace dans t1 toutes les occurrences de la sous-chaîne t2 par la chaîne t3

```
exemple d'utilisation :  txt ← 'interro d''algo'
                        remplacerOcc(txt, ' d''algo', ' de math')
                        ecrire(txt)                                # affiche : interro de math
```

- *estNumerique*(*t* : *texte*) :

cette procédure détermine si la chaîne `t` possède un contenu numérique

[illegible]

- *trouvez vous-même l'une ou l'autre fonction et/ou procédure supplémentaire(s) qui vous semble intéressante(s) pour le traitement des chaînes de caractères*

## 2) Cette version (provisoire) d'un tri de tableau a été donnée à une époque :

```

Algorithme tri

Constante tInf = 1, tSup = 200      # définition de la taille physique du tableau
Type typIndicesP = tInf .. tSup    # intervalle des indices physiques
Type typIndicesR = 0 .. tSup       # intervalle des indices réels
Variable t : Tableau[typIndicesP] de Entier
    bInf, bSup : typIndicesR      # variables indiquant les bornes réelles
    estTrié : Logique              # état du tableau

Procédure parcoursEchange :
variable : ech : entier            # variable pour l'échange
    indice: typIndicesP           # indice d'itérateur
Début
    estTrié ← vrai                # l'optimisme habituel
    Pour indice de bInf à bSup - 1
        Si t[indice] > t[indice + 1] Alors      # manifestement le tableau n'est pas ordonné
            ech ← t[indice]                    # on fait l'échange ...
            t[indice] ← t[indice + 1]
            t[indice + 1] ← ech
        estTrié ← faux                    # ... et nécessité d'un nouveau parcours
    finSi
finPour
Fin

Procédure triBulle :
Début
    Répéter
        parcoursEchange
    Jusque estTrié
Fin

Procédure chargerTableau :
Début
    ...
Fin

Procédure afficherTableau :
Début
    ...
Fin

Début                                # tri
    chargerTableau
    triBulle
    afficherTableau
Fin.

```

*En supposant qu'un algorithme déclare les trois variables suivantes :*

```

Constante tInf = 1, tSup = 200      # définition de la taille physique du tableau
Type typIndicesP = tInf .. tSup    # intervalle des indices physiques
Type tabEntier = Tableau[typIndicesP] de Entier
Variable t1, t2, t3 : tabEntier

```

- rédiger (avec un ensemble cohérent de fonctions et procédures, de leur(s) paramètre(s), et en utilisant correctement sur le mode de passage de ceux-ci) une version de l'algorithme permettant de charger, trier et afficher chacun de ces tableaux à son tour (ou à la demande) ; (si nécessaire, modifier également le bloc déclaratif des données)*
- rédigez une fonction `chercheSeq()` et une fonction `chercheDicho()` permettant d'effectuer (séquentiellement pour la première, dichotomiquement pour la seconde) la recherche d'une valeur (entière) particulière dans un de ces trois tableaux (valeur retournée par ces fonctions : vrai/faux)*
- en utilisant la structure d'enregistrement, proposez une implémentation (données et outils algorithmiques) qui allège la programmation du chargement/tri/affichage/recherche*

### 3) Travail de groupe (semaine 8, novembre, année 2010-2011)

*Rappelez-vous ... l'année dernière, à votre demande ☺, le Père Noël vous avait apporté une petite station météorologique que vous avez installée dans votre jardin, et – respectant la résolution que vous aviez prise – vous effectuez depuis lors et chaque matin un relevé de température, de pression atmosphérique et de pluviosité, mesures soigneusement encodées et conservées informatiquement ... (via un programme rédigé par vos soins 🐣 en langage Pascal) ... vous avez ainsi pu établir toutes sortes de données statistiques (moyennes, min/max, ...) à partir des tableaux mensuels patiemment collectés ...*

*Hasard ou conséquence ? On vous propose aujourd'hui d'effectuer un stage au service informatique du SRCICDUGRW (Service Régional de Centralisation Informatique des Collectes de Données sans doute Utiles à la Gestion de la Région Wallonne) et plus spécifiquement dans son département DGOSPGCVF (Département Général Opérationnel de Surveillance, de Prévision et de Gestion de Crise des Voies Fluviales) afin de participer à un vaste projet de collecte de données relatives aux cours d'eau traversant la Wallonie.*

#### CONTEXTE GÉNÉRAL

*Le saviez-vous ? il y a des fleuves, des rivières et des canaux qui traversent la Wallonie, et pour se faire une idée de l'état de chacun et pouvoir prévoir une évolution à court terme (p.ex. les douze prochaines heures) et à moyen terme (p.ex. deux à cinq jours), il faut rassembler des données telles que : la pluviosité (eh oui, ce sont les pluies qui alimentent les cours d'eau ! les précipitations sont mesurées en mm, ce qui correspond à des litres/m<sup>2</sup>), le débit (volume d'eau par unité de temps, exprimé en m<sup>3</sup>/sec) et la hauteur d'eau (ou niveau, exprimé en m).*

*Pour ce faire, le long de chaque cours d'eau, sont disposées de petites stations automatiques de mesure (alimentées par une batterie rechargée en journée par un panneau solaire) disposant chacune d'un pluviomètre, d'un limnigraphe (appareil mesurant la hauteur d'eau) et d'un débitmètre; chacune dispose d'un modem et est ainsi raccordée au réseau téléphonique, ce qui assure la transmission de ses mesures à un serveur et permet une télésurveillance de l'état des appareils*

*Il y a actuellement 230 stations de ce type (et il est prévu à terme de deux ans de couvrir l'ensemble du territoire avec un maximum de 300 stations)*



#### STATIONS

*Pour identifier chaque station, on souhaite*

- *lui affecter un code numérique (entier) unique (p.ex. 1043 pour la station de "Wavre")*
- *lui donner un nom (p.ex. de la ville ou du lieu-dit où elle est placée), lui aussi unique (quand il y a plusieurs stations dans une même ville sur un même cours d'eau, on lève l'ambiguïté en précisant le lieu, p.ex. "Walcourt-gare" et "Walcourt-seuil")*
- *spécifier à quel cours d'eau elle est associée (une station ne gère qu'un seul cours d'eau) (p.ex. "Meuse", "Ourthe", "Canal Albert")*
- *enregistrer ses coordonnées, en longitude et latitude (degrés, minutes et secondes d'arc)*
- *enregistrer sa date d'installation (année, mois, jour)*
- *enregistrer la date de la dernière intervention du service de maintenance (année, mois, jour)*

## MESURES

*Chaque station effectue automatiquement quatre mesures par jour (à 00h00, 06h00, 12h00 et 18h00)*

*Chaque prise de mesure comporte trois valeurs (somme de pluviosité sur les 6 dernières heures, débit et hauteur instantanés, dans cet ordre), (p.ex. 23.5 142.5 1.82 NB : avec 1 décimale pour la pluviosité et le débit, 2 décimales pour la hauteur)*

*Chacune des quatre prises, précédée du code de la station, de la date (année, mois, jour) et du numéro de prise (dans cet ordre), est aussitôt transmise par le modem à un programme de collecte et de mémorisation. (p.ex. 1043 2010 11 27 1 23.5 142.5 1.82)*

*Ce programme devra conserver l'ensemble des mesures (considérez que la prise de mesures a commencé en 2001 et que le programme doit encore tenir cinq ans avant une prochaine réinformatisation)*

*Ce programme devra également permettre toutes les opérations, par exemple,*

- *la gestion des stations (ajout, modification, suppression)*
- *la gestion des mesures (modification de mesures erronées, encodage manuel en cas de panne de station, ...)*
- *la mise à disponibilité des mesures (listes diverses à des fins statistiques, ...)*

## VOTRE MISSION

### *1. structure de données (ADT)*

- a) *proposez une structure de données pour représenter le concept de station; veillez à la structurer de telle sorte que l'encodage puisse s'effectuer de manière simple à travers des fonctions et ou procédures réutilisables constituant son API*
- b) *proposez une structure de données pour représenter le parc de stations disponibles*
- c) *proposez une structure de données pour représenter le concept de mesure transmise par la station vers le serveur*
- d) *proposez une structure de données pour mémoriser l'ensemble des mesures pour la durée de vie prévue pour le programme*

### *2. interface de programmation (API)*

- a) *sur les structures de données que vous avez proposées en 1a) et 1b), rédigez un ensemble de fonctions et de procédures paramétrées permettant d'ajouter une nouvelle station au parc existant (ces données doivent évidemment être soigneusement validées, et veillez également à tout (?) prévoir ... p.ex. essayer d'ajouter une station existant déjà !)*
- b) *sur les structures de données que vous avez proposées en 1c) et 1d) écrivez une procédure (ou une fonction) paramétrée permettant l'enregistrement d'une mesure transmise par une station (aucune validation n'est à prévoir)*
- c) *rédigez une procédure paramétrée permettant de lister les stations dans l'ordre alphabétique des cours d'eau (en majeur) et des villes (en mineur)*
- d) *rédigez une procédure paramétrée permettant de lister les stations dans l'ordre chronologique de leur installation*



- e) écrivez une fonction paramétrée qui renvoie le numéro unique d'une station (sur base de son nom p.ex. "Wavre" et du cours d'eau qu'elle gère p.ex. "Dyle")
- f) en utilisant cette fonction, écrivez une procédure paramétrée qui affiche chronologiquement tous les relevés de débit d'une station donnée, pour une année, un mois et une prise données (p.ex. mars 2008 à 12h00)
- g) en utilisant cette fonction, écrivez une autre fonction qui renvoie la moyenne de pluviosité pour un mois, une année et une station données
- h) en utilisant cette nouvelle fonction, écrivez une procédure paramétrée qui affiche la moyenne de pluviosité pour une station, une année et un mois donnés
- i) écrivez une procédure paramétrée permettant de connaître les cinq dernières opérations de maintenance sur le parc de stations

### CONSIGNES

1. constituez des groupes de 3 étudiant(e)s décidé(e)s à collaborer
2. il ne faut pas rédiger de séquence principale ! ceci est un exercice portant uniquement sur la conception d'une structure de données et des procédures et fonctions associées qui serviront d'outils algorithmiques ensuite, lors de la réalisation de l'application proprement dite; en conséquence, il n'y a donc aucune variable globale à prévoir (mais les constantes éventuelles font partie de la conception des données)
3. vous devez évidemment répondre d'abord aux quatre points de la question 1)
4. vous devez écrire l'en-tête de tous les outils algorithmiques (procédures et fonctions); pour rappel, l'en-tête est la ligne de déclaration reprenant le nom de l'outil, le nom et le type de chaque paramètre formel, et pour les fonctions, le type de la valeur de retour
5. vous devez répondre aux questions 2a), 2b) et 2e), 2g)
6. vous devez répondre à l'une des deux questions 2c) ou 2d) au choix
7. vous devez répondre aux questions 2e) et 2g)
8. vous pouvez répondre à l'une ou l'autre autre question à votre choix (primes)

### CONSEILS

En algorithmique, puis en programmation, la clé du succès est souvent la simplicité associée à du simple bon sens ... ne concevez pas des 'usines à gaz'<sup>71</sup> ...

... une décomposition – tant du côté des données que du côté des outils – est donc vivement recommandée; meilleure sera la structure de données, plus aisée sera sa programmation !

... la simplicité, c'est également la clarté et la précision des noms, tant du côté des données que du côté des outils ...

... et la présence de commentaires bien 'dosés', ni trop peu, ni trop

... sans oublier la qualité et la présentation du pseudo-code lui-même

<sup>71</sup> [http://fr.wikipedia.org/wiki/Usine\\_à\\_gaz](http://fr.wikipedia.org/wiki/Usine_à_gaz)

## ANNEXE



Le département DGOSPGCVF du SRCICDUGRW est évidemment une pure invention du rédacteur de cet exercice !

Plus sérieusement ... depuis plus de 10 ans, au sein du SETHY (Service d'Études Hydrologiques de la Région Wallonne), le service Hydrologie - dont il faut saluer le travail des ingénieurs, techniciens et informaticiens - a installé et connecté 230 stations de mesure et publie sur son site internet diverses pages reprenant des données globales, moyennes ou détaillées pour chacun des bassins hydrographiques et leurs cours d'eau

<http://voies-hydrauliques.wallonie.be/opencms/opencms/fr/hydro/bulhyd.html>

On y trouve les grandes tendances des principaux cours d'eau (moyenne quotidienne)

Portail Wallonie | Accueil - Plan du site - Contacts

Direction générale opérationnelle de la Mobilité et des Voies hydrauliques  
"Les Voies hydrauliques, les voies du développement durable..."

Vous êtes ici : Accueil > Hydrologie > Bulletin hydrologique

**Rubriques**

- Les voies d'eau
- Promotion et intermodalité
- Transport par eau
- Tourisme et loisirs
- Hydrologie**
  - Objectifs et méthodes
  - Données actuelles / InfoCrue
  - Données archivées et statistiques
- Infos

**Accès direct**

- Statistiques de navigation
- Les avis à la batellerie / R.I.S.
- Le journal de la batellerie
- Le bulletin hydrologique
- Etat des eaux / Infocruce
- Les aides à la batellerie
- Les horaires des ouvrages d'art

**Outils**

- Qui sommes-nous ?
- Nous contacter
- Outils de recherche
- Liens/adresses utiles

**HYDROLOGIE 27 nov. 2010**

Version imprimable

Les débits en m<sup>3</sup>/s sont des données **provisoires** et correspondent à la moyenne de la journée.  
Les débits d'aujourd'hui correspondent à la moyenne des 6 premières heures.

**Etat des eaux sur les voies navigables**

Rivière	Lieu	Date : 25/11	26/11	27/11
MEUSE	HEER-AGIMONT	145	133	126
MEUSE	WAULSORT	146	134	130
MEUSE	AMPSIN-NEUVILLE	195	187	178
MEUSE	WISE	205	190	161
SAMBRE	SOLRE	13.2	11.3	11.7
SAMBRE	NAMUR	28.9	25.2	26
DENDRE CANALISEE	BILHEE Barrage-Ecl	2.21	2.1	2.02
ESCAUT	TOURNAI	27.7	28	21.8

**Etat des eaux sur les autres rivières**

Rivière	Lieu	Date : 25/11	26/11	27/11
SEMOIS	MEMBRE Pont	28.6	26.4	25.1
VIROIN	TREIGNES	6	5.33	5.05
LESSE	GENDRON	14.4	13.4	12.6
OURTHE	NISRAMONT	8.8	7.43	7.35
OURTHE	HAMOIR-TABREUX	17.4	16.5	15.5
OURTHE	ANGLEUR	58.4	50.5	46.9
AMBLEVE	MARTINRIVE	26.7	18.5	16.8
HAINE	BOUSSOIT	1.93	1.73	1.61

**Avertissement** : Les données publiées ci-dessus sont des valeurs provisoires, susceptibles d'être modifiées après contrôle. Elles ne donnent qu'une indication des tendances hydrologiques en temps réel et ne peuvent être utilisées à d'autres fins.

Via une carte (muette) interactive (de la Wallonie) ... et si on en profitait pour essayer d'y nommer les cours d'eau et d'y placer les principales villes ??? ... on a accès pour la sélection opérée ...

Wallonie | Vous êtes ici : Accueil > Hydrologie > Données actuelles / InfoCrue > Etat des eaux / InfoCrue

**Rubriques**

- Les voies d'eau
- Promotion et intermodalité
- Transport par eau
- Tourisme et loisirs
- Hydrologie**
  - Objectifs et méthodes
  - Données actuelles / InfoCrue
  - Etat des eaux / InfoCrue
  - Bulletin
  - Situation générale
  - Bilans trimestriels
  - Données archivées et statistiques
- Infos

**Accès direct**

- Statistiques de navigation
- Les avis à la batellerie / R.I.S.
- Le journal de la batellerie
- Le bulletin hydrologique
- Etat des eaux / Infocruce
- Les aides à la batellerie
- Les horaires des ouvrages d'art

**Outils**

- Qui sommes-nous ?
- Nous contacter
- Outils de recherche

**Etat des eaux en Région wallonne**

Provinces  
Cours d'eau

Phase

- Alerte de crue
- Pré-alerte de crue
- Normale
- Etiage

Pour une information plus détaillée :

- choisir une zone sur la carte
- choisir un cours d'eau
- choisir une commune

Comment utiliser cette page ?

Que faire en cas de crue ?

... à quatre mesures quotidiennes (par tranche de 6 heures) des trois grandeurs ...

Portail Wallonie | Accueil - Plan du site - Contacts

Direction générale opérationnelle de la Mobilité et des Voies hydrauliques  
"Les Voies hydrauliques, les voies du développement durable..."

Vous êtes ici : Accueil > Hydrologie > Données actuelles / InfoCruce > Etat des eaux / InfoCruce

**Rubriques**

- Les voies d'eau
- Promotion et intermodalité
- Transport par eau
- Tourisme et loisirs
- Hydrologie**
  - Objectifs et méthodes
  - Données actuelles / InfoCruce
  - Etat des eaux / InfoCruce**
    - Bulletin
    - Situation générale
    - Bilans trimestriels
    - Données archivées et statistiques
- Infos

**Accès direct**

- Statistiques de navigation
- Les avis à la batellerie / R.I.S.
- Le journal de la batellerie
- Le bulletin hydrologique
- Etat des eaux / Infocruce
- Les aides à la batellerie
- Les horaires des ouvrages d'art

**Outils**

- Qui sommes-nous ?
- Nous contacter
- Outils de recherche
- Liens/adresses utiles

Secteur : HAUTE MEUSE  
HAUTE MEUSE : Situation normale

**Avertissement** : Les données publiées ci-dessus sont non contrôlées. La fréquence de mise à jour est variable en fonction des stations et de la situation hydrologique.

		Hauteurs (m)					
Station	Rivière	Phase	27/11 - 6h	27/11 - 12h	27/11 - 18h	27/11 - 24h	28/11 - 6h
PROFONDEVILLE	MEUSE		1,46	1,44	1,44	1,44	1,41
LUSTIN	MEUSE		1,49	1,48	1,47	1,46	1,44
DINANT	MEUSE		1,16	1,14	1,14	1,13	1,11
ANSEREMME Monia	MEUSE		1,39	1,38	1,38	1,37	1,35
YVOIR	BOCQ		0,40	0,39	0,39	0,39	0,39
WARNANT	MOLIGNEE		0,49	0,49	0,49	0,49	0,49
HASTIERE	HERMETON		0,38	0,37	0,37	0,36	0,36
FELENNE	HOUILLE		0,59	0,59	0,58	0,58	0,57
CHOOZ	MEUSE		0,86	0,87	0,86	0,81	0,83

		Débits (m³/sec)					
Station	Rivière	Phase	27/11 - 6h	27/11 - 12h	27/11 - 18h	27/11 - 24h	28/11 - 6h
YVOIR	BOCQ		1,5	1,5	1,5	1,5	1,5
WARNANT	MOLIGNEE		1,3	1,2	1,2	1,2	1,2
CHOOZ	MEUSE		125,0	127,2	125,0	118,0	121,1

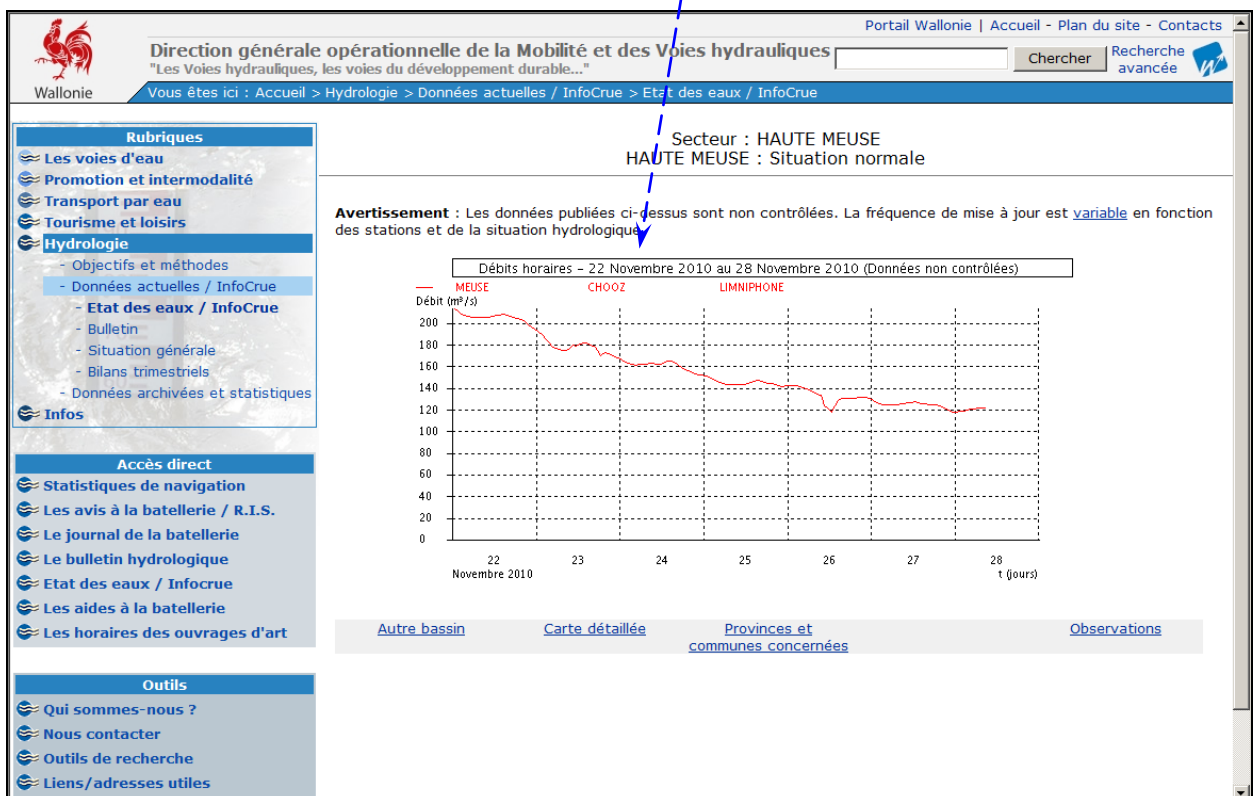
		Pluies (mm) - somme sur 6 heures					
Station	Rivière	Phase	27/11 - 6h	27/11 - 12h	27/11 - 18h	27/11 - 24h	28/11 - 6h
SAINT-GERARD	(BURNOT)		0,0	0,0	0,0	0,0	0,0
CRUPET	(BOCQ)		0,0	0,0	0,2	0,0	0,0
CINEY	(BOCQ)		0,0	0,0	0,8	0,0	0,0
FLORENNES	(MOLIGNEE)		0,0	0,3	0,1	0,0	0,0
GEDINNE	(HOUILLE)		0,0	0,0	0,1	0,0	0,0

[Autre bassin](#) [Carte détaillée](#) [Provinces et communes concernées](#)

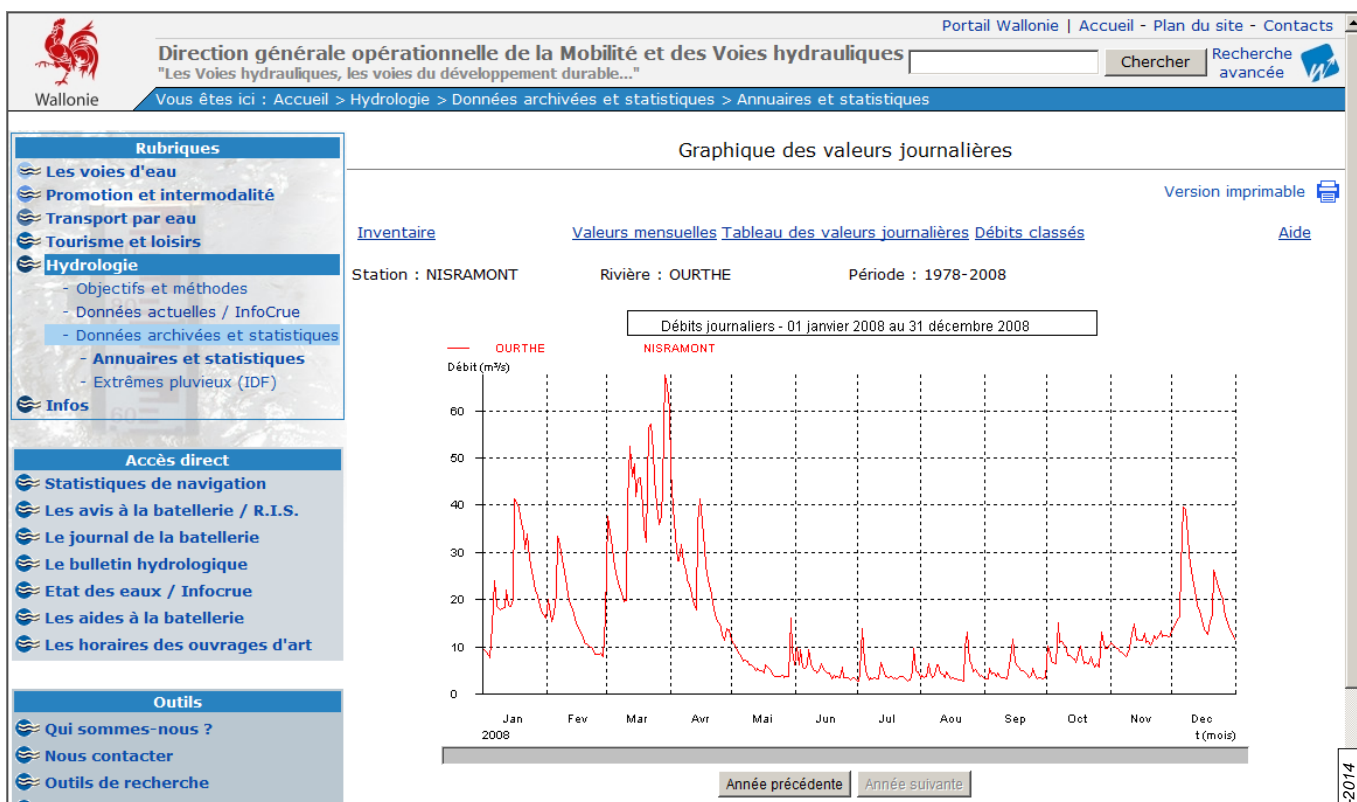
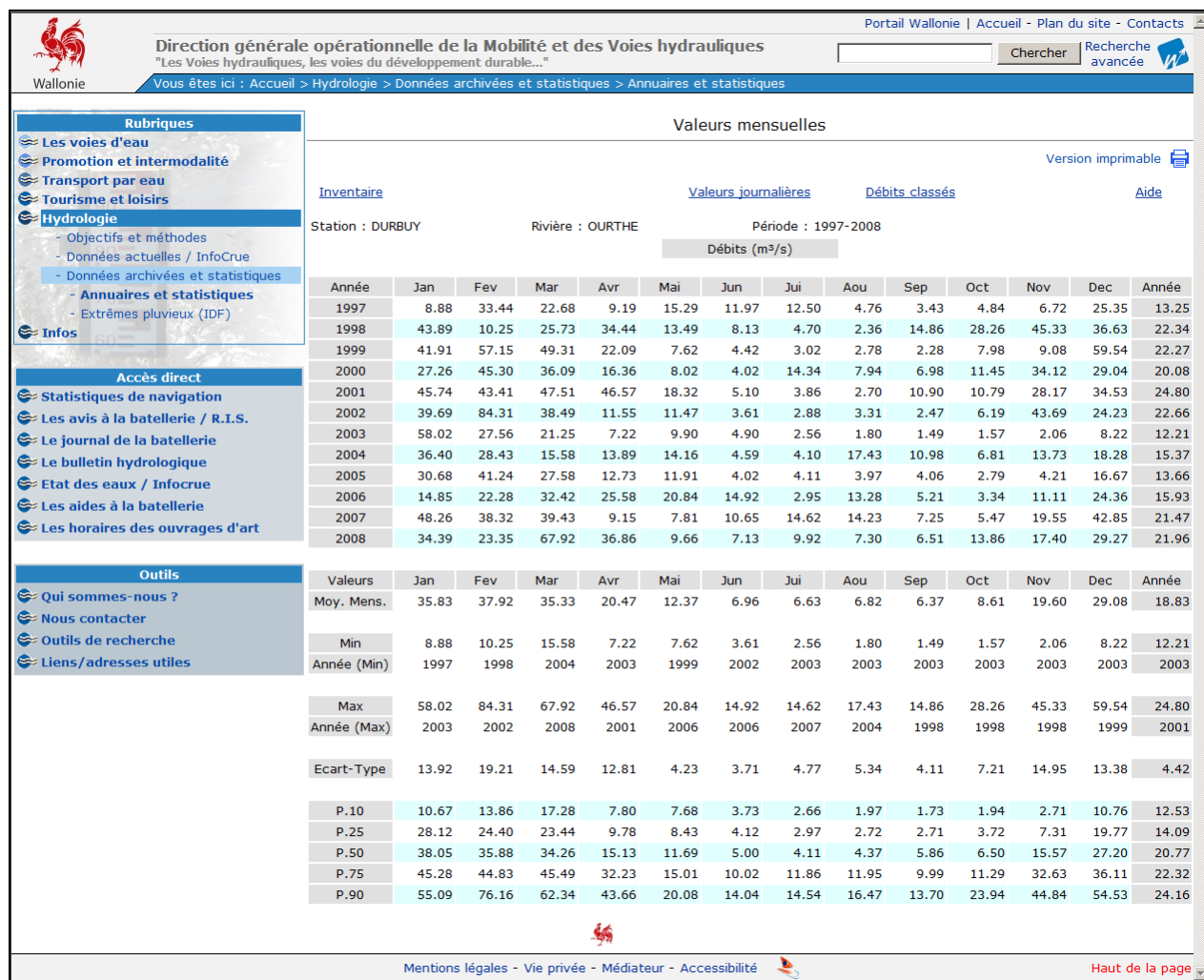
Mentions légales - Vie privée - Médiateur - Accessibilité

Haut de la page

... et notamment aux débits et hauteurs qui sont disponibles et visualisables graphiquement quasiment 'en temps réel' (les stations envoient une mesure par heure aujourd'hui)



Enfin, de très nombreux tableaux et graphiques récapitulatifs sont consultables en ligne



4) Pour informatiser un service commercial, on utilise les déclarations suivantes :

- *les produits :*

```

Constante maxProduits = 4000           # nombre maximum de produits
Type tProduit = Enregistrement         # définition d'un produit
    code      : Texte                  # code unique, référence, p.ex. 'P012'
    libellé   : Texte                  # nom du produit, p.ex. 'marteau 2kg'
    prix      : Réel                   # prix unitaire du produit, p.ex. 12.5
Fin
Type indProduit = 0..maxProduits       # un type sûrement utile pour la recherche
Type tProduits = Enregistrement        # ensemble de produits
    nbProd    : indProduit             # nombre de produits encodés
    tbProd    : Tableau[maxProduits] de tProduit # tableau de produits
Fin
variable catalogue : tProduits         # concrètement, un catalogue de produits

```

**savez-vous :** (les ??? ci-dessous indiquent que vous devez déclarer des paramètres et leur mode de passage)

- 1) *en utilisant les déclarations ci-dessus et la variable catalogue (mais sans écrire de procédure ou de fonction, il s'agit ici de maîtriser l'accès aux données via les structures) :*
  - initialiser le catalogue ?
  - donner le prix du cinquième produit du catalogue ?
  - donner le libellé du dernier produit du catalogue ?
  - indiquer le nombre de produits encodés ?
- 2) *écrire une procédure ajouterProduit(???) qui permette d'encoder un nouveau produit depuis le clavier (saisie d'un code et d'un libellé) et de l'ajouter dans le catalogue ? (seule validation à prévoir pour l'instant : reste-t-il de la place dans le catalogue ?)*
- 3) *écrire une fonction chercheCodeProduit(???) : indProduit qui permette de vérifier qu'un produit (sur base de son code, passé comme paramètre) existe ou non dans le catalogue ? en cas de succès, la fonction retourne l'indice du produit dans le tableau; en cas d'échec, la fonction retourne 0*
- 4) *modifier la procédure ajouterProduit ci-dessus afin de vérifier - à l'aide de la fonction chercheCodeProduit - que le nouveau produit n'a pas déjà été encodé dans le catalogue ? s'il existe, la procédure doit refuser l'ajout de ce produit*
- 5) *écrire une procédure trierCatalogue(???) permettant de trier le catalogue par ordre croissant des codes ?*
- 6) *modifier la procédure ajouterProduit ci-dessus afin qu'après ajout d'un nouveau produit, le catalogue reste trié par ordre croissant des codes ?*
- 7) *écrire une fonction libProduit(???) : Texte qui – par recherche séquentielle dans un tableau trié - renvoie le libellé d'un produit (sur base de son code, passé comme paramètre) et renvoie 'indisponible' sinon ? (on suppose le catalogue trié au moment où elle s'exécute)*
- 8) *écrire une procédure afficherCatalogueCode(???) qui affiche l'ensemble des produits du catalogue dans l'ordre croissant des codes ? (on suppose le catalogue trié au moment où elle s'exécute)*
- 9) *écrire une procédure afficherCatalogueLib(???) qui affiche l'ensemble des produits du catalogue dans l'ordre alphabétique des libellés, sans pour autant modifier le catalogue qui doit rester trié dans l'ordre croissant des codes ? (indication/rappel : si on passe par valeur un paramètre à une procédure, celle-ci peut modifier sans problème ce paramètre)*

Si vous ne maîtrisez pas les procédures et fonctions paramétrées, vous pouvez toujours rédiger une version non paramétrée de ces outils, en travaillant directement sur les variables globales ...

... et si le type tProduits vous perturbe, décomposez-le en deux variables globales nbProd et tbProd et utilisez-les directement (dans une version non paramétrée des outils procéduraux)

... et si c'est le type tProduit qui vous perturbe, rédigez une version multi-tableaux du problème

5) Pour informatiser un service commercial - suite de l'énoncé Révisions précédent dans lequel les produits sont définis - on utilise les déclarations suivantes :

• *les fournisseurs :*

```

Constante maxFourn = 100           # nombre maximum de fournisseurs
Constante maxLivr = 100           # nombre maximum de produits livrés par fournisseur
Type iFourn = 0..maxFourn
Type iLivr = 0..maxLivr
Type tFourn = Enregistrement      # définition d'un fournisseur
  référence : Texte               # un code unique d'identification (p.ex. 'F0025')
  nom       : Texte               # nom du fournisseur, p.ex. 'hilti'
  adresse   : Enregistrement      # adresse
    rue      : Texte              # rue
    codePostal : 1000..9900        # code postal
    ville     : Texte              # localité
  Fin
  mail      : Texte               # adresse mail
  téléphone : Texte               # téléphone
  livraisons : Enregistrement     # produits livrés par le fournisseur
    indLivr : 0..maxLivr           # nombre de produits encodés
    tbLivr : Tableau[maxLivr] de Texte # tableau des codes de ces produits
  Fin
Fin
Type tabFourn = Enregistrement    # ensemble de fournisseurs
  nbFourn : 0.. maxFourn           # nombre de fournisseurs encodés
  tbFourn : tableau[maxFourn] de tFourn # tableau de fournisseurs
Fin ;
Variable fournisseurs : tabFourn  # concrètement, une liste de fournisseurs

```

**savez-vous :** (les ??? ci-dessous indiquent que vous devez déclarer des paramètres et leur mode de passage)

- 1) *en utilisant les déclarations ci-dessus et la variable fournisseurs (mais sans écrire de procédure ou de fonction, il s'agit ici de maîtriser l'accès aux données via les structures) :*
  - donner le nombre de fournisseurs ?
  - donner le nom du cinquième fournisseur ?
  - donner le code postal du dernier fournisseur ?
  - indiquer le nombre de produits livrés par le premier fournisseur ?
  - donner le prix du troisième produit livré par le dernier fournisseur ?
- 2) *écrire une procédure ajouterFournisseur(???) qui permette d'encoder un nouveau fournisseur depuis le clavier (par saisie des informations signalétiques) et de l'ajouter dans la liste ? lors de cette création, veillez à ne pas oublier de vérifier qu'il reste de la place dans le tableau de fournisseurs et d'initialiser la liste des produits livrés par ce fournisseur !*
- 3) *écrire une fonction chercheFournisseur(???) : iFourn qui permette de vérifier qu'un fournisseur (sur base de sa référence uniquement) existe dans la liste ? si la recherche est fructueuse, la fonction renvoie l'indice du fournisseur dans la table des fournisseurs, sinon, elle renvoie 0*
- 4) *modifier la procédure ajouterFournisseur(???) afin de vérifier - à l'aide de la fonction - que le nouveau fournisseur n'a pas déjà été encodé dans la liste ?*
- 5) *écrire une fonction mailFournisseur(???) : Texte qui renvoie l'adresse mail d'un fournisseur dont on connaît que le téléphone; la fonction renvoie 'n/a' si un tel fournisseur ne figure pas dans la liste et renvoie '---' si le fournisseur trouvé n'a pas d'adresse mail ?*
- 6) *écrire une procédure ajouterFournProd(???) qui permette (pour un fournisseur dont on fournit la référence) d'ajouter un produit (dont on fournit le code) à la liste de ses livraisons ? veillez à vérifier que le fournisseur existe dans la liste, que le produit fait partie du catalogue, et qu'il est possible de l'ajouter ?*
- 7) *écrire une procédure listerFournProd(???) qui affiche – pour un fournisseur dont on connaît la référence – la liste alphabétique des produits qu'il fournit*
- 8) *écrire une procédure listerProdFourn(???) qui affiche – pour un produit dont on connaît le code – la liste alphabétique des fournisseurs qui fournissent ce produit*



- 6) Note : cet exercice faisait partie de la liste des propositions d'énoncés pour l'examen de janvier 2009 ; comme il n'a pas été retenu, en voici une version 'de révision' plus compète (beaucoup plus de questions, couvrant plus largement l'ensemble du cours ...)

### Tournois de cartes

*Une maison de retraite nous a sollicités pour informatiser la gestion des tournois de jeu de cartes qu'elle organise pour ses pensionnaires ; après analyse, nous avons décidé de l'implémentation des données suivante :*

- *Les pensionnaires : (!! il ne peut en exister deux possédant le même nom et prénom !!)*

```

Constante maxPens = 40                # nombre maximum de pensionnaires
Type tPersonne = Enregistrement      # définition d'un pensionnaire
  nom : Texte                        # nom du pensionnaire
  prénom : Texte                    # prénom du pensionnaire
Fin
Type indPens = 0.. maxPens
Type tPensionnaires = Enregistrement # ensemble de pensionnaires
  nbPens : indPens                  # nombre de pensionnaires encodés
  tabPens : Tableau[maxPens] de tPersonne # tableau de pensionnaires
Fin

```

- *Les équipes : elles sont constituées de deux pensionnaires différents (!! un pensionnaire ne peut appartenir qu'à une seule équipe !!)*

```

Constante maxEquipes = 10             # nombre maximum d'équipes
Type tEquipe = Enregistrement         # définition d'une équipe
  nomEquipe : Texte                  # nom de l'équipe
  joueurs : Tableau[2] de indPens    # référence aux deux joueurs ! indices
Fin
Type indEquipe = 0..maxEquipes
Type tEquipes = Enregistrement        # ensemble d'équipes
  nbEquipes : indEquipe              # nombre d'équipes encodées
  tabEquipes : tableau[maxEquipes] de tEquipe # tableau d'équipes
Fin

```

- *Les matches : ils voient s'affronter deux équipes différentes (!! chaque équipe dispute plusieurs matches, mais deux équipes différentes ne s'affrontent qu'une seule fois par tournoi !!)*

```

Constante maxMatches = 20            # nombre maximum de matches
Type tMatch = Enregistrement         # définition d'un match
  equipe1, equipe2 : indEquipe        # référence aux deux équipes ! indices
  score1, score2 : Entier             # le score de chacune
Fin
Type indMatch = 0..maxMatches
Type tMatches = Enregistrement       # ensemble de matches
  nbMatches : indMatch               # nombre de matches encodés
  tabMatches : tableau[maxMatches] de tMatch # tableau de matches
Fin

```

*Pour une première version de la gestion des tournois, nous avons créé les variables globales suivantes :*

```

Variable résidents : tPensionnaires
  équipes : tEquipes
  tournoi : tMatches

```

*Nous vous sollicitons à notre tour pour écrire un certain nombre d'outils procéduraux sur cette implémentation*

1. *préliminaire ... en utilisant seulement les déclarations ci-dessus et les variables globales (mais sans pour autant écrire de procédure ou de fonction, c'est un simple d'exercice d'accès aux données), savez-vous :*
  - *indiquer le nombre de pensionnaires encodés ?*
  - *indiquer le nombre d'équipes encodées ?*
  - *en supposant deux équipes déjà constituées et encodées, en créer une troisième dont le nom est 'les casquettes' et les références des joueurs sont les pensionnaires n° 25 et 33 ?*
  - *initialiser un tournoi ?*
  - *en supposant le tournoi terminé (c-à-d toutes les données encodées), donner les noms des équipes participant au sixième match et leurs scores respectifs ?*

**savez-vous écrire :** (les ??? ci-dessous indiquent que vous devez déclarer des paramètres et leur mode de passage)

2. *une procédure ajouterPensionnaire(???) qui permet de saisir – au clavier – le nom et le prénom d'un(e) nouveau(elle) pensionnaire et de créer ce(tte) pensionnaire.*
  - *pensez aux validations nécessaires ...*
  - *cette procédure devrait utiliser une fonction (que vous devriez écrire) cherchePensionnaire(???) : indPens pour vérifier que la personne n'a pas déjà été encodée (si la personne existe, la fonction renvoie sa 'place' dans la liste des pensionnaires, sinon elle renvoie 0)*
3. *une fonction nombrePersonnes(???) : indPens renvoyant le nombre actuel de pensionnaires*
4. *une procédure initialiserTournoi(???) qui – avant l'organisation d'un nouveau tournoi – supprime les données (équipes et matches) du tournoi précédent*
5. *une fonction nombreEquipes(???) : indEquipe renvoyant le nombre actuel d'équipes constituées*
6. *une procédure triAlphaEquipes(???) qui trie la table des équipes alphabétiquement sur le nom de celles-ci*
7. *une procédure listeAlphaEquipes(???) qui – après exécution de la procédure précédente – affiche la liste alphabétique et la constitution des équipes (nom de l'équipe, nom et prénom de chaque joueur)*
8. *une procédure ajouterEquipe(???) qui permet d'enregistrer une nouvelle équipe, à partir de la saisie au clavier du nom de l'équipe et des noms et prénoms des deux joueurs.*
  - *cette procédure devrait utiliser une fonction (que vous deviez écrire) existeEquipe(???) : indEquipe pour vérifier qu'une équipe du même nom n'a pas déjà été encodée (si l'équipe existe, la fonction renvoie son indice dans la table des équipes, sinon elle renvoie 0)*
  - *cette procédure devrait utiliser la fonction cherchePensionnaire() écrite précédemment ...*
  - *cette procédure devrait également utiliser une fonction (que vous devriez écrire) existeJoueur(???) : Logique pour vérifier qu'aucun des deux partenaires ne fait pas déjà partie d'une autre équipe*
9. *une fonction nombreMatches(???) : indMatch renvoyant le nombre actuel de matches enregistrés*
10. *une procédure créerMatch(???) qui – sur base des noms des deux équipes concernées - crée un nouveau match et enregistre le résultat obtenu par ces équipes;*
  - *il est vivement conseillé d'utiliser la fonction existeEquipe() écrite précédemment*
  - *veillez également à écrire une fonction existeMatch(???) :Logique afin de vérifier que ce match n'a pas été déjà enregistré*
11. *une fonction equipeGagnante(???) : Texte renvoyant le nom de l'équipe (et les noms et prénoms des joueurs qui la constituent) qui a gagné le plus de matches (une équipe gagne un match si son score est strictement supérieur au score de l'équipe adverse, on ne tient donc pas compte des matches nuls)*

Si vous ne maîtrisez pas les procédures et fonctions paramétrées, vous pouvez toujours rédiger une version non paramétrée de ces outils, en travaillant directement sur les variables globales ...

Si les types abstraits vous perturbent, décomposez-les en variables globales ... (plus lourd, ☹)

... ou rédigez une version multi-tableaux du problème (tout à fait lourd, ça ! ☹☹)



## 7. EXAMENS DES ANNÉES PRÉCÉDENTES

### 1) Examen de première session : 2008-2009

*Les baccalauréats de Promotion Sociale (en cours du soir) proposent aux étudiants de s'inscrire à une ou plusieurs UF (Unités de Formation, des cours en quelque sorte) 'à la carte'.*

*Le 'catalogue' de ces formations peut être décrit à l'aide des structures de données suivantes :*

```
Type typCursus = Enregistrement      # définition d'une unité de formation (UF)
  codeForm      : Texte                # son code p.ex. 'elmar'
  libelléForm   : Texte                # son intitulé p.ex. 'éléments de marketing'
  nombrePériodes : 10..200            # son nombre de périodes/année, p.ex. 80
Fin

Constante maxForm = 100                # une limite supérieure
Type typFormations = Enregistrement    # l'ensemble des unités de formations (UFs)
  nbFormations  : 0..maxForm           # leur nombre, p.ex. 43
  tabFormations : Tableau[maxForm] de typCursus # le tableau de ces formations
Fin

Variable listeUFs : typFormations      # concrètement : le 'catalogue', la liste des formations
```

*Au moment de leur inscription, les étudiants reçoivent chacun un matricule et leur(s) inscription(s) aux unités de formation de leur choix sont mémorisées dans une structure de données telle que celle-ci :*

```
Constante maxUFs = 10                # une limite supérieure au nombre d'UFs
Type typEtudiant = Enregistrement    # définition d'un étudiant
  matricule : Texte                  # son matricule, p.ex. 'PS080125'
  nom       : Texte                  # son nom, p.ex. 'MARTIN'
  tabUFs    : = Enregistrement      # ses inscriptions (son cursus)
  nbInscr   : 0..maxUFs              # combien il suit d'UFs, p.ex. 9
  tabInscr  : Tableau[maxUFs] de Texte # un tableau des codes de ces UF
Fin

Constante maxEtu = 999                # une limite supérieure au nombre d'étudiants
Type typPopuleEtu = Enregistrement    # définition de la population étudiante
  nbEtu      : 0..maxEtu              # le nombre d'étudiants déjà inscrits
  tabEtu     : Tableau[maxEtu] de typEtudiant # un tableau de ces étudiants
Fin

Variable Étudiants : typPopuleEtu      # concrètement, la 'population', la liste des étudiante
```

- L'analyste qui vous a précédé sur ce projet a déjà rédigé des outils logiciels, dont la fonction suivante, qui peut s'avérer utile : si un étudiant de matricule donné existe, la fonction renvoie sa 'place' dans la liste des étudiants, elle renvoie 0 sinon

```
Fonction existeEtu(matr : Texte ; pEtu : typPopuleEtu) : 0..maxEtu
Variable ind : 0..maxEtu, estTrouvé : Logique
Debut
  ind ← 0
  estTrouvé ← faux
  tantQue non (estTrouvé OU ind = pEtu.nbEtu)
    ind ← ind + 1
    estTrouvé ← (pEtu.tabEtu[ind].matricule = matr)
  finTQ
  Si estTrouvé alors renvoie(ind)
    sinon renvoie(0)
  finSi
Fin
```

dans ce qui suit, les ??? présents dans la signature des procédures et/ou des fonctions indiquent que vous devez spécifier des paramètres, leur type et leur mode de passage

- a) rédigez une fonction `existeUF(???)` : `0..maxUFs` permettant de vérifier – sur base d'un code fourni - qu'une UF figure ou non dans la liste des formations ; (si elle existe, sa 'place' dans la liste est renvoyée par la fonction, sinon la fonction renvoie 0)
  
- b) rédigez une fonction `nombreEtu(???)` : ??? renvoyant le nombre d'étudiants inscrits
  
- c) rédigez une procédure `ajouterEtu(???)` permettant de saisir au clavier les caractéristiques (matricule, nom) d'un nouvel étudiant , puis d'ajouter cet étudiant; effectuez tous les contrôles qui vous semblent nécessaires à cet ajout et initialisez correctement son cursus
  
- d) rédigez une procédure `afficherUFs(???)` permettant d'afficher la liste des formations existantes (code, libellé, périodes)
  
- e) rédigez une procédure `trierUFs(???)` assurant que le catalogue des formations soit dans l'ordre croissant des codes
  
- f) rédigez une procédure `trierEtuAlpha(???)` permettant d'afficher la liste des étudiants (matricule, nom) dans l'ordre alphabétique des noms ; il vous est demandé de constituer pour ce faire un index séparé sur les noms d'étudiants

## 2) Examen de première session : 2009-2010

Contexte général

*Vous êtes un grand fan de cinéma, et en quelques années, vous avez accumulé chez vous - au fil de vos achats ☺ ou de vos copies ☺ - une vaste collection de DVD que vous rangez dans plusieurs étagères.*

*Il y en a tellement que vous êtes devenu incapable de retrouver facilement un film : vous devez parcourir ces étagères, planche par planche, DVD par DVD pour enfin 🕵️ trouver l'objet de votre désir.*

*A première vue, vous en avez actuellement plusieurs centaines et vous envisagez d'en posséder environ deux mille à moyen terme ...*

Vos intentions ...

*Maintenant que vous savez programmer, c'est décidé : vous allez informatiser votre DVD-thèque ! sans doute avec tableau(x) en attendant mieux ...*

*Une brève analyse de vos besoins vous conduit à la décision suivante : pour chaque DVD, il faudrait enregistrer :*

- le titre (chaîne de caractères en texte libre)
- le genre (policier, drame, espionnage, animation, science-fiction, horreur ... : chaîne de caractères en texte libre)
- l'année de sortie du film (par défaut 9999 si inconnue)
- la durée (en minutes, entre 40 et 180)
- l'emplacement du rangement, sous forme d'une chaîne de caractères : n°étagère – n°planche – n°disque (les étagères sont numérotées de 1 à 9; dans chaque étagère, les planches sont numérotées de 1 à 9 et sur chaque planche, les DVD sont numérotés de 01 à 99, ce qui vous permet de stocker plus de 8000 DVD !)

*Pour vous faciliter la tâche de programmation, vous décidez également que toutes les chaînes de caractères (titre et genre) seront encodées avec soin et en minuscules ... et que la chaîne de caractères représentant l'emplacement du rangement est valide (aucune validation n'est à prévoir donc sur ces chaînes)*

*Avec un collègue grand amateur de programmation Pascal, vous décidez d'une structure de données commune et compatible pseudo-code / Pascal, ce qui donne*

```

Constante maxDVD = 8000
Type tDVD = Enregistrement
  titre : Texte
  genre : Texte
  année : Entier
  durée : 40..180
  place : Texte
Fin
Type tTablDVD = Tableau[maxDVD] de tDVD
Variable nbDVD : 0..maxDVD
      dvdTheque : typTablDVD
  
```

```

const maxDVD = 8000;
type tDVD = Record
  titre : String;
  genre : String;
  année : Integer;
  durée : 40..180;
  place : String
end;
type tTablDVD = array[1..maxDVD] of tDVD;
var nbDVD : 0..maxDVD;
      dvdTheque : typTablDVD;
  
```

Exemple de données

titre	genre	année	durée	rangement étagère-planche-disque
fantasia	animation	9999	125	2-3-11
l'armée des 12 singes	aventure	1995	129	1-2-06
scream 1	horreur	1996	111	2-3-04
hostel 1	horreur	2005	94	1-3-02
the constant gardener	drame	2005	125	1-3-10
pole express	animation	2004	159	2-1-04
...	...	...	...	...

## Fonctionnalités

*De la sorte, vous espérez – notamment grâce aux possibilités qu'offrent les tris - pouvoir*

- *trouver immédiatement l'emplacement de rangement d'un film dont vous connaissez le titre (ou savoir que vous ne le possédez pas)*
- *connaître le nombre de films en votre possession*
- *avoir la liste alphabétique des films d'un genre particulier (horreur, p.ex.)*
- *etc ...*

*Il vous est demandé ici :*

1. *sur base des structures de données qui vous sont fournies à la page précédente, de compléter le bloc déclaratif des données avec les constantes, types et/ou variables supplémentaires que vous jugerez nécessaires*

2. *de rédiger l'algorithme général d'un (futur) programme :*

2.1. *articulé autour d'un menu, qui, après avoir correctement initialisé la DVD-thèque, propose (sous forme d'appels de procédures) les options suivantes :*

- *ajouter un nouveau film (détails ci-dessous)*
- *afficher l'emplacement de rangement d'un film (idem)*
- *afficher le nombre total de films (idem)*
- *afficher la liste alphabétique des films (idem)*
- *quitter l'application*

2.2. *procédure d'ajout de film; spécifications :*

*(attention 1 : vous pouvez posséder plusieurs films portant le même titre, mais uniquement à condition qu'ils aient des années de sortie différentes !)*

*on encode d'abord le titre (en minuscules) du film et son année (valide) de sortie; s'il n'existe pas dans la DVD-thèque (cfr. ci-dessous), on encode et valide le reste des informations et on ajoute ce film, sinon ... on espère un message d'erreur clair*

*(attention 2 : l'ajout de nouveaux films s'effectue toujours derrière tous les films existants, pour autant bien sûr qu'il reste de la place dans la DVD-thèque.)*

*pour vérifier que le nouveau DVD existe déjà ou pas dans la DVD-thèque, il vous est demandé d'écrire et d'utiliser une fonction de recherche dont la signature est*

*fonction rechercheDVD(titr : xxxx; ann : xxxx) : xxxx;*

*où les xxxx représentent des déclarations de types appropriées que vous devez rédiger; en cas de succès, la fonction renvoie l'indice du film dans le tableau dvdTheque et renvoie 0 sinon*

2.3. *procédure de consultation d'emplacement de rangement; spécifications :*

*on encode uniquement le titre (en minuscules); si un(des) DVD existe(nt) dans la DVD-thèque, la procédure en fournit la liste : une ligne par film reprenant genre, année, durée et emplacement*

2.4. *procédure d'affichage du nombre de films; spécifications :*

*on espère l'affichage d'une ligne comportant le message 'vous possédez xxx films' ou 'vous ne possédez aucun film !'*

2.5. *procédure d'affichage de la liste alphabétique des films; spécifications :*

*on attend – si elle existe – la liste des films (une ligne par film, reprenant titre, genre, année, durée et emplacement de rangement) triée dans l'ordre alphabétique des titres*

## 3) Examen de première session : 2010-2011

**Question 1.**

Soient d'une part les structures abstraites suivantes :

```

Constante maxOuvr = 1000           # nombre maximum d'ouvrages dans la librairie
Type TOuvrage = Enregistrement
    code :       Texte              # référence unique, p.ex. "INF009"
    titre :      Texte              # titre de l'ouvrage, p.ex. "Algorithmique"
    auteur :     Texte              # auteur de l'ouvrage, p.ex. "Wirth N."
    prix :       Réel               # prix unitaire, p.ex. 12.50€
    qtéStock :   Entier             # quantité de cet ouvrage en stock, p.ex. 9
finEnr
Type TCatalogue = Enregistrement   # catalogue/stock : tableau 'partiel'
    tbOuvrages : Tableau[maxOuvr] de TOuvrage      # tableau physique
    nbOuvrages : Entier                           # nombre d'ouvrages du catalogue
finEnr

```

Soit d'autre part un algorithme comportant les déclarations concrètes suivantes :

```

Variable catal1, catal2 : TCatalogue
Variable val : Réel

```

... et contenant les instructions suivantes :

```

# valeur du catalogue/stock catal2
val ← valStock(catal2)

# ajout d'un nouvel ouvrage dans le catalogue/stock catal1
addCatal(catal1, "MAT026", "Trigonométrie", 11.95, 20)

```

Rédigez (déclaration complète + code) les outils algorithmiques paramétrés suivants :

- la fonction `valStock(?)` : ? retournant la valeur (en €) d'un stock
- la procédure `addCatal(?)` permettant d'ajouter un nouvel ouvrage dans un catalogue; vous devez écrire et utiliser une fonction vérifiant que cet ajout est possible ... (
- un outil procédural permettant de déterminer si un ouvrage dont on connaît le titre figure dans le stock et en combien d'exemplaires; on suppose ici que tous les ouvrages possèdent un titre différent

NB : dans ce qui précède, le ? signifie que la déclaration des paramètres (nombre, nom, type et mode de passage) est à votre charge ...

**Question 2.**

Une société commerciale de vente par téléphone (téléachat)

- dispose d'une équipe de (maximum 20) vendeurs caractérisés chacun par un code (unique), un nom, un prénom, un sexe, une date de naissance, une date d'engagement dans la société
- chaque jour, chaque vendeur essaie – par prospection téléphonique - de réaliser des ventes; pour chacune de ses ventes réalisées (un maximum de 30 par jour), on souhaite conserver le numéro appelé, l'heure, et le montant (€) de la vente
- à la fin de chaque journée, l'ensemble des données relatives aux ventes du jour est collecté, synthétisé (on ne conserve que le total réalisé par vendeur) et enregistré sur un serveur; cet ensemble est ensuite réinitialisé en prévision du travail du lendemain

*Proposez (rédigez) une structure abstraite permettant de représenter*

- le concept de vendeur*
- le concept d'équipe de vendeurs*
- le concept de vente réalisée*
- le concept d'ensemble de ventes du jour d'un vendeur*
- le concept d'historique des ventes*

### Question 3.

*Dans le tiroir du bureau de l'informaticien que vous remplacez, vous trouvez une feuille rédigée comme ceci :*

```

Constante maxElem = 1000

Type TElem = Enregistrement
    a : Texte
    b : Texte
    x : Réel
finEnr

Type TtbElem = Enregistrement
    tbElem : Tableau[maxElem] de TElem
    nbElem : Entier
finEnr

Procédure tri(ref t : TtbElem) :
# tri de t en majeur sur x décroissant et sur (a, b) en mineur croissant
Variable indice, nbEch : Entier; txt : Texte; rel : Réel
Début
    Répéter
        nbEch ← 0
        Pour indice ← 1 à t.nbElem - 1 Faire
            Si t.tbElem[indice + 1].x > t.tbElem[indice].x Alors
                nbEch ← 1
            Sinon Si t.tbElem[indice + 1].x = t.tbElem[indice].x Alors
                Si t.tbElem[indice + 1].a < t.tbElem[indice].a Alors
                    nbEch ← 1
                Sinon Si t.tbElem[indice + 1].a = t.tbElem[indice].a Alors
                    Si t.tbElem[indice + 1].b < t.tbElem[indice].b Alors
                        nbEch ← 1
                    finSi
                finSi
            finSi
        finPr
        Si nbEch > 0 Alors
            txt ← t.tbElem[indice].a
            t.tbElem[indice].a ← t.tbElem[indice + 1].a
            t.tbElem[indice + 1].a ← txt
            txt ← t.tbElem[indice].b
            t.tbElem[indice].b ← t.tbElem[indice + 1].b
            t.tbElem[indice + 1].b ← txt
            rel ← t.tbElem[indice].x
            t.tbElem[indice].x ← t.tbElem[indice + 1].x
            t.tbElem[indice + 1].x ← rel
        finSi
    Jusqu'à nbEch = 0
Fin

```

*Allégez et récrivez cet algorithme en utilisant (déclarant, rédigeant, invoquant)*

- une fonction comparatrice paramétrée pour déterminer si l'échange est nécessaire (aucune contrainte particulière ne vous est imposée, vous avez toute liberté pour ce faire)*
- une procédure paramétrée pour réaliser cet échange (idem)*

## 4) Examen de première session : 2011-2012

**A. Première partie**

Pour les étudiants en année d'études terminale, un établissement d'enseignement supérieur organise les travaux de fin d'études (TFEs) à l'aide de la structure de données suivantes<sup>72</sup> :

▪ structures abstraites : types 'utilisateur' (ADT)

```

Constante maxEtu = 60           # nombre maximum d'étudiants
      maxProfs = 20             # nombre maximum d'enseignants
      maxTfes = 60             # nombre maximum de TFEs différents

Type TDate = Enregistrement    # définition abstraite d'une date
  année : Entier
  mois  : Entier
  jour  : Entier
finEnr

Type TEtudiant = Enregistrement # définition abstraite d'un étudiant
  matricule : Texte[8]          # matricule (unique), p.ex. "HE201129"
  nom       : Texte[50]
  prénom    : Texte[25]
  sexe      : Caractère         # valide si 'm' ou 'f'
  classe    : Texte[4]          # groupe, p.ex. "3TL2"
  bisseur   : Logique           # vrai ou faux
finEnr

Type TEnseignant = Enregistrement # définition abstraite d'un enseignant
  référence : Texte[5]          # mnémonique (unique), p.ex. "Ch.L.", "MN.V."
  nom       : Texte[50]
  prénom    : Texte[25]
  sexe      : Caractère         # valide si 'm' ou 'f'
finEnr

Type TTfe = Enregistrement      # définition abstraite d'un tfe
  numéro    : Entier            # référence (unique) du tfe, p.ex. 227
  catégorie : Texte[4]          # domaine, p.ex. "prog", "elec", "web", "netw"
  titre     : Texte[255]        # titre du tfe
  confidentiel : Logique        # vrai ou faux
  nbEtu     : Entier            # nombre d'étudiants associés à ce tfe
  tbEtu     : Tableau[3] de Texte[8] # matricules des étudiants coauteurs
  rapporteur : Texte[5]         # référence (mnémonique) d'un enseignant
  nbLecteurs : Entier           # nombre de lecteurs
  tbLecteurs : Tableau[2] de Texte[5] # référence (mnémonique) d'enseignant(s)
  approbation : TDate           # date d'approbation par le rapporteur
finEnr

```

## ▪ trois définitions abstraites de tableaux 'partiels'

```

Type TpEtudiants = Enregistrement # ensemble (liste) d'étudiants
  nbEtudiants : Entier
  tbEtudiants : Tableau[maxEtu] de TEtudiant
finEnr

Type TpEnseignants = Enregistrement # ensemble (liste) d'enseignants
  nbEnseignants : Entier
  tbEnseignants : Tableau[maxProfs] de TEnseignant
finEnr

Type TpTfes = Enregistrement # ensemble (liste) de TFEs
  nbTfes : Entier
  tbTfes : Tableau[maxTfes] de TTfe
finEnr

```

<sup>72</sup> fournies ici à titre indicatif; vous pouvez – à condition de le justifier par un commentaire – y apporter les modifications conceptuelles ou techniques qui vous semblent utiles

▪ quelques détails et contraintes :

- chaque étudiant possède un matricule unique qui l'identifie sans ambiguïté
- chaque enseignant est identifié de manière unique et sans ambiguïté par un mnémonique constitué à partir des initiales de son nom et de son prénom (p.ex. "Ch.L.")
- un TFE peut être réalisé par un à trois étudiants (différents) maximum, qui sont coauteurs du travail
- tout étudiant ne peut participer qu'à un seul TFE
- un TFE est réalisé dans une des cinq catégories (spécialisations) suivantes : "prog" (programmation/développement), "elec" (électronique etc...), "web" (internet etc...), "netw" (réseaux etc...), "bdd" (bases de données etc...)
- durant sa réalisation, un TFE est suivi par un enseignant (le 'rapporteur') et - pour la défense en jury – par aucun, un ou deux autres enseignants supplémentaires (les 'lecteurs'); tous ces enseignants sont spécialisés dans un des domaines cités ci-dessus (celui du TFE) et participeront à un jury dans cette spécialité
- tous les enseignants peuvent être rapporteurs ou lecteurs de TFEs; mais pour un TFE donné, un enseignant ne peut pas occuper à la fois le rôle de rapporteur et de lecteur
- dans chacune des structures concrètes (tableaux partiels), les données sont encodées sans ordre particulier (en fait, simplement dans l'ordre chronologique de leur arrivée)

Question 1

*en utilisant la structure de données de la page précédente, créez trois variables globales permettant de mémoriser respectivement un ensemble d'étudiants, d'enseignants et de TFEs ; quels sont alors les identificateurs correspondant aux informations suivantes ?*

- a) le nombre de TFEs introduits et encodés
- b) le mnémonique du dernier enseignant (encodé)
- c) le caractère confidentiel du second TFE (encodé)
- d) la classe (groupe) à laquelle appartient le premier étudiant (encodé)
- e) le nombre d'étudiants qui sont coauteurs du premier TFE (encodé)
- f) le nombre d'enseignants 'lecteurs' associés à ce même TFE

Question 2

*en utilisant la structure de données de la page précédente:*

- a) rédigez une procédure<sup>73</sup> `listerTfes` fournissant la liste des TFEs (numéro, catégorie, titre, confidentiel, rapporteur, approbation) triée en majeur sur catégorie et en mineur sur titre
- b) rédigez deux fonctions paramétrées<sup>74</sup> `identiteEtu(...)` et `identiteProf(...)` renvoyant respectivement la concaténation de l'initiale du prénom et du nom d'un étudiant dont on connaît le matricule et la concaténation de l'initiale du prénom et du nom d'un enseignant dont on connaît la référence (mnémonique); en cas d'erreur, ces fonctions doivent renvoyer une chaîne vide

<sup>73</sup> vous pouvez écrire ici si vous le jugez utile une ou plusieurs autres procédures et/ou fonctions; vous avez le droit également d'utiliser soit directement vos variables concrètes (globales), soit de passer (mais correctement) des paramètres

<sup>74</sup> obligatoirement cette fois ..., seuls les noms de ces fonctions figurent ci-dessus, le reste est à votre charge



- c) en invoquant ces fonctions, rédigez une nouvelle version<sup>75</sup> de la procédure écrite en a) ci-dessus, en privilégiant pour chaque TFE la présentation suivante :

TFE : numéro    catégorie    titre  
 étudiants : identité du(des) étudiant(s) coauteur(s)  
 rapporteur : identité du rapporteur  
 lecteur(s) : identité du(des) lecteur(s)

exemple

TFE : 227    web    Site de e-Commerce pour collectionneurs de timbres  
 étudiants : M.DEMARET - N. de VILLERS  
 rapporteur : MN.V.  
 lecteur(s) : Ch.L. - Y.B.

## B. Seconde partie

*On vous demande d'imaginer la structure de données nécessaire à la constitution et à l'organisation des jurys de TFEs d'une part, et de la gestion de l'horaire de passage des étudiants pour la défense orale du travail d'autre part ...*

*La problématique en est la suivante*

- on constitue des jurys de TFEs (numérotés de 1 à 5), un pour chacune des spécialités (domaines, catégories) évoquées dans la question précédente
  - chaque jury se réunit à une date précise (unique) dans un local de classe (unique); si des jurys distincts opèrent en parallèle (à la même date), ce doit évidemment être dans des locaux distincts ...
  - chaque jury est composé d'un ensemble d'enseignants (rapporteurs, lecteurs ou invités) spécialisés dans le domaine ...
- une fois les jurys constitués, on construit ensuite l'horaire de passage :
  - il s'agit d'une défense par travail (TFE), ce qui signifie que les étudiants coauteurs se présentent et sont interrogés ensemble (le temps de défense est plus ou moins proportionnel au nombre d'étudiants, de l'ordre de 20 min par étudiant)
  - il suffit donc d'associer à chaque travail (TFE) le numéro de son jury et l'heure de passage (entre 8h30 et 17h00)
  - d'autre part, après chaque défense, une petite délibération interne a lieu et le jury accorde au travail deux cotes (/50), la première pour la qualité technique de la réalisation, la seconde pour la qualité de la présentation orale ; ces deux cotes doivent être mémorisées

### Question 3

- a) rédigez les déclarations de types abstraits (ADT) et les variables concrètes correspondant à la description d'un jury, à la constitution des différents jurys et à l'horaire de passage de défense
- b) en imaginant l'encodage au moyen d'un ensemble de procédures/fonctions constituant l'API de cette ADT, listez (sans pour autant en rédiger le code) de manière claire (et en mentionnant sur quelles données ils portent) l'ensemble de tous les contrôles et validations qu'il faudrait idéalement effectuer pour que l'ensemble des données soit cohérent (p.ex. une validation a été mentionnée ci-dessus : des jurys en parallèle doivent opérer dans des locaux distincts)

*avec votre structure de données, rédigez l'algorithme fournissant – pour chacun des jurys – l'ordre de passage des étudiants dans l'ordre chronologique*

<sup>75</sup> vous avez ici encore toute liberté d'action pour atteindre l'objectif fixé

## 5) Examen de première session : 2012-2013

- Pour gérer le projet d'algorithmique de fin de semestre, l'enseignant concerné a réalisé une structure de données abstraite (ADT) sur base des considérations suivantes :
  - la population étudiante est répartie en classes (60 étudiants maximum)
  - pour réaliser le projet, les étudiants constituent des équipes (de 2 à 4 étudiants maximum) à l'intérieur d'une même classe (pas d'équipe avec des étudiants de classes différentes)
  - un étudiant ne peut appartenir qu'à une seule équipe
  - évidemment, certains étudiants décident de ne pas faire le projet ☹
  - à l'échéance de la constitution des équipes, celles-ci sont enregistrées (encodées)
  - le projet comporte deux phases : la réalisation d'une ADT, puis la rédaction de son API ; ces phases ont chacune une échéance et les travaux sont soumis par les équipes sur eCampus
  - chaque phase est cotée séparément
  - évidemment, certaines équipes, bien que constituées et enregistrées, renoncent en cours de route, soit dès l'ADT, soit pour l'API ☹

### Définition du concept d'étudiant et de classe

```
Type TEtudiant = Enregistrement      # définition abstraite du type étudiant
    matricule : Texte                # matricule identifiant (unique) de l'étudiant
    nom       : Texte                # nom de famille de l'étudiant
    prénom    : Texte                # prénom de l'étudiant
    sexe      : Caractère            # sexe (m/f) de l'étudiant
finEnr

constante maxEtu = 60                # nombre maximum d'étudiants dans une classe
type TpClasse = Enregistrement      # classe abstraite = tableau partiel d'étudiants
    tbEtu : Tableau[maxClasse + 1] de TEtudiant  # tableau physique d'étudiants
    nbEtu : Entier                      # nombre d'étudiants
finEnr
```

### Définition du concept d'équipe et de projet

```
constante tailleEq = 4 # nombre maximum d'étudiants dans une équipe de projet
type TpEquipe = Enregistrement      # une équipe = un tableau partiel de participants
    tbEquipe : Tableau[tailleEq + 1] de Texte  # tableau physique de matricules
    nbEquipe : Entier                      # nombre d'étudiants dans
l'équipe
finEnr

type TProjet = Enregistrement      # définition abstraite d'un projet
    equipe : TpEquipe              # une équipe (un tableau partiel d'étudiants)
    remisADT : Logique              # remise de la partie ADT sur eCampus (oui/non)
    coteADT : Réel                  # cote obtenue (sur 20) si remis
    remisAPI : Logique              # remise de la partie API sur eCampus (oui/non)
    coteAPI : Réel                  # cote obtenue (sur 20) si remis
finEnr
```

### Définition du concept d'ensemble de projets d'une classe

```
constante maxProjets = 30          # nombre maximum de projets (60 étud. 2 min. par projet)
type TpProjets = Enregistrement    # liste des projets = tableau partiel
    tbProjets : Tableau[maxProjets] de TProjet  # tableau physique de projets
    nbProjets : Entier                  # nombre de projets
finEnr
```

### Exemple de variables concrètes

```
variable clas1TL, clas1TM : TpClasse      # deux classes ...
    proj1TL, proj1TM : TpProjets          # ... et leurs projets
```

- 1.1. complétez les appels de l'invocation de la procédure standard `écrire(???)` pour afficher :
- le matricule du dernier étudiant de 1TM
  - le nombre d'étudiants de la première équipe de 1TL et leurs matricules respectifs
  - le nombre maximum d'étudiants pouvant constituer une équipe

- 1.2. rédigez le code d'une fonction permettant de connaître le nombre d'équipes qu'une classe a constituées

*sa signature doit être* `Fonction nbEquipes(c : TpProjet) : Entier`

*exemple d'invocation :* `écrire(nbEquipes(proj1TM))`

- 1.3. rédigez la signature (le paramétrage est donc à votre charge) et le code d'une fonction permettant de connaître le nombre de phases ADT du projet postées sur eCampus par une classe particulière

*son nom doit être fonction* `nbADT`

*exemple d'invocation :* `écrire(nbADT(???))` # ??? = selon votre paramétrage

- 1.4. écrivez une fonction permettant de déterminer si un étudiant – sur base de son matricule – fait ou non partie d'une équipe

*signature :* `Fonction estEnEquipe(e : TEtudiant, t : TpProjet) : Logique`

*exemple d'invocation*

```
variable e : TEtudiant
lire(e.matricule)
si estEnEquipe(e, proj1TL) Alors écrire("ok") Sinon écrire("nok") Fin-
Si
```

- 1.5. à l'aide de cette fonction, dressez la liste des étudiants d'une classe qui ne font pas partie d'une équipe via une procédure dont la signature (à compléter) est

*signature :* `Procédure étudiantsSansEquipe(???)` # ??? = déclarez les paramètres

*exemple d'invocation :* `étudiantsSansEquipe(clas1TL, proj1TL)`

- 1.6. rédigez une fonction comparatrice permettant d'ordonner alphabétiquement deux étudiants sur base de leur identité (nom + prénom) ; cette fonction renverra -1 si l'identité de e1 est alphabétiquement 'avant' celle de e2, 0 si les deux identités de e1 et e2 sont identiques et +1 si celle de e1 est 'après' celle de e2

*signature :* `Fonction compareEtu(e1, e2 : TEtudiant) : Entier # -1, 0, 1`

- 1.7. déclarez (signature complète) et rédigez une fonction utilisant cette fonction comparatrice pour obtenir le matricule d'un étudiant dont on connaît l'identité

- 1.8. déclarez (signature complète) et rédigez une procédure utilisant cette fonction comparatrice pour afficher la population d'une classe par ordre alphabétique des étudiants (nom en majeur et prénom en mineur)

2. Une bibliothèque communale propose à ses adhérents des prêts d'ouvrages selon les règles suivantes

- *un adhérent est une personne qui s'est inscrite à la bibliothèque, qui a reçu lors de cette inscription (et à vie) un numéro d'adhérent ; pour sa gestion interne, la bibliothèque enregistre pour chaque adhérent ses nom, prénom, sexe, date de naissance, adresse complète, téléphone fixe et gsm, adresse mail, date d'inscription*
- *régulièrement, la bibliothèque établit différentes listes de ses adhérents (par numéro d'adhérent, par ordre alphabétique d'identité, par ordre chronologique des dates d'inscription, etc ...)*
- *chaque ouvrage possède un numéro unique, un type (livre, revue, dictionnaire, ...), un titre et un emplacement (n° armoire et n° de planche dans l'armoire)*
- *un catalogue des ouvrages disponibles est régulièrement mis à jour et est proposé à la consultation aux adhérents*
- *un prêt est un ouvrage qui est emporté par un adhérent ; un ouvrage ne peut être conservé qu'un maximum de 3 semaines (passé ce délai, un rappel doit pouvoir être envoyé à l'adhérent) ; quand l'ouvrage rentre, le prêt est supprimé ; un adhérent ne peut pas avoir plus de 5 prêts en cours*

Proposez (rédigez) une structure abstraite (ADT) permettant de représenter

- c) *le concept d'adhérent et de liste des adhérents de la bibliothèque*
- d) *le concept d'ouvrage et de catalogue d'ouvrages*
- e) *le concept de prêt et d'ensemble des prêts en cours de la bibliothèque*

## SOURCES

WIRTH N., Systematic programming: An introduction, Prentice-Hall, 1973

KNUTH D.E., Sorting and Searching (vol. 3 of The Art of Computer Programming), AddisonWesley, 1973

WIRTH N, Algoritms + Data Structures = Programs, Prentice-Hall, 1976

ARSAC J., Premières leçons de programmation, Nathan, 1980

MEYER B. & BAUDOIN C., Méthodes de programmation, Eyrolles, 1980

RAYMON F.H., Informatique – Programmation, CNAM, Masson, 1982

ARSAC J., Les bases de la programmation, Dunod, 1983

MEYER B., Conception et programmation orientées objet, Eyrolles, 2000

DUCHÂTEAU Ch., Images pour programmer FUNDP Namur, 1990 - 2002

WARIN B., L'algorithmique, votre passeport informatique pour la programmation, Ellipses, 2002

Cours d'Algorithmique, Université de Paris XI (IUT d'Orsay), 2003

COLLECTIF, Exercices & Problèmes d'algorithmique, Dunod, 2010