Images pour programmer

2ème partie

Charles Duchâteau

Département
Éducation
et Technologie

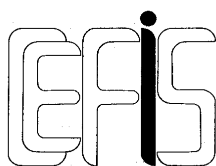
- Les tableaux
- L'approche descendante
- Les limites de l'approche descendante

Images pour programmer
2ème partie

Charles Duchâteau

5.78

Mai 2002



Centre pour la Formation à l'Informatique dans le Secondaire

Avant-propos

Voici, après plus de 10 ans, la deuxième partie de "*Images pour programmer*". Et même si le format ou la mise en page en est différente, c'est toujours la même volonté de vulgariser et de rendre accessible qui est à l'œuvre dans les pages qui suivent.

10 ans et le paysage dans lequel s'inscrit la programmation a été profondément bouleversé : en un mot, l'informatique a cédé le devant de la scène aux "technologies de l'information et de la communication" (TIC). Lorsque la première partie de "Images pour programmer" est parue, on en était toujours, en ce qui concerne la micro-informatique, à MS-DOS et à (ce qu'on n'appelait pas encore) des interfaces de type texte. Depuis, Windows est apparu dans le monde PC et simultanément a commencé la fuite en avant des versions successives, de plus en plus complexes et puissantes, de moins en moins fiables aussi. La valse ininterrompue des variantes successives des logiciels d'application a bien entendu accompagné celle des systèmes d'exploitations.

Et puis, il y a dix ans, Internet n'existait pas : pas de WEB, pas d'e-mail, ... Pas de multimédia non plus, pas de DVD, pas d'appareil photo numérique, encore moins de caméra,...

J'ai, à de multiples reprises, évoqué l'univers des usages des technologies et le cauchemar qu'elles représentent pour la plupart de leurs utilisateurs. Je n'y reviendrai pas ici, même si un certain désenchantement pour ces "technologies" n'est pas étranger à un regain d'intérêt personnel pour la programmation et son apprentissage. Cela fait du bien de retrouver des activités où l'on a l'impression d'utiliser son cerveau...

Mais ce long délai m'a permis aussi de prendre du recul vis à vis de l'approche adoptée dans la première partie de "*Images pour programmer*", de mesurer ce qui était adéquat, de regretter ce qui l'était moins, en un mot, d'approfondir la réflexion didactique à propos de l'apprentissage de l'algorithmique et de la programmation.

1. Des constats

Permettez-moi d'abord de faire aveu de deux ou trois malaises qui continuent à me poursuivre au fil des pages qui suivent.

1.1 Assis entre deux chaises

Dans ce deuxième volume, comme dans le premier, le propos continue à louvoyer, à mi-chemin entre l'algorithmique pure et dure et des détails de "cuisine technique" propre à un langage (ici Pascal). C'est que le choix est clairement fait d'un point de départ qui soit toujours une tâche "complète" à programmer. Ainsi, dans les pages suivantes, on rencontrera des tâches comme la détermination de la fréquences des diverses lettres dans un texte, le tirage du Lotto, la confection d'une calculatrice en chiffres romains, l'écriture de nombres en toutes lettres,...

Les énoncés proposés ne commencent donc jamais par "étant donné un tableau indexé par..." ou "Proposer un algorithme qui...". A chaque fois, on part d'une tâche, on la précise et on va jusqu'à l'écriture d'un programme en Pascal, qui soigne suffisamment les entrées-sorties et où il faut bien se préoccuper de détails "techniques", par exemple de lecture ou d'affichage.

C'est que, clairement, le parti est pris d'avancer à la fois dans la connaissance de concepts et

d'outils généraux de la programmation impérative, mais aussi d'approfondir (sans excès) la connaissance d'un langage et même parfois d'une implémentation particulière d'un langage.

1.2 On apprend la programmation en programmant

Voilà bien l'un des drames de l'enseignement de la programmation (comme de beaucoup d'autres d'ailleurs) : quoi que fasse l'enseignant, quelle que soit l'approche qu'il propose, quelles que soient les métaphores et les images qu'il utilise pour aider à la compréhension, bref quel que soit l'art d'enseigner qu'il tente de déployer ou la réflexion didactique qui sous-tende son propos, la programmation ne s'apprend, ni en lisant les livres qui en parlent, ni en suivant des exposés qui en traitent : il n'y a qu'une manière de maîtriser la programmation et c'est d'écrire des programmes...

Voilà qui doit rendre l'enseignant bien modeste : quoi qu'il fasse ou qu'il écrive, ce sont celles et ceux à qui il s'adresse qui ont les clés de leur propre apprentissage. Et la maîtrise de la programmation ne s'acquiert qu'en programmant...

1.3 Le langage : quand et comment ?

Dans le présent volume, comme dans le premier, le parti est clairement de distinguer la conception et l'écriture de l'algorithme (de la "marche à suivre"), au sein de l'étape du "Comment faire faire ?", de son expression dans le langage Pascal, objet de l'étape "Comment dire ?".

Il faut se rendre compte qu'il y a quinze ans, lorsque j'ai écrit "Programmer !" d'abord, "Images pour programmer" ensuite, les langages, quels qu'ils soient, avaient mauvaise presse. On sortait d'une époque où "apprentissage de la programmation" rimait souvent avec "détails syntaxiques à propos d'un langage". Pour se démarquer de cette fâcheuse tendance, il fallait donc "remettre le langage à sa place", celle de simple mode d'expression d'un algorithme conçu par ailleurs, en utilisant un autre mode d'expression (GNS, pseudo-code, langage de description d'algorithme,...) moins suspect de dérive syntaxique. En fait, on remplaçait un "vrai" langage (un langage de programmation) par un "faux" langage (non interprétable par l'ordinateur); mais un langage pour s'adresser à l'ordinateur et exprimer les algorithmes, il en faut toujours un...

On relira avec profit le chapitre 1 des "Première leçons de programmation" de Jacques Arsac, pour se replonger dans ce qui était, en 1980, la perception des dérives antérieures de l'enseignement de la programmation ("*Ayant confondu programmation et langages, on s'était contenté d'enseigner les langages*.").

Il reste quelques arguments qui continuent à plaider, surtout **au tout début de l'apprentissage** de la programmation, pour une expression dans un "pseudo- langage" plutôt que directement dans un vrai langage de programmation, fut-ce Pascal :

- Les contraintes syntaxiques propres aux "vrais" langages de programmation n'ont pas à venir interférer avec la phase de conception de la "marche à suivre" en ajoutant à la difficulté de "penser juste" une couche "écrire sans faute"; il ne faut pas mélanger la perception de la portée d'un "Tant que..." ou de la nécessité de telle variable avec la multiplication des begin ou des points-virgules...
- L'expression sous forme de pseudo-code (ou de tout autre moyen d'expression des algorithmes) permet, au début en tous cas, une traduction ultérieure dans le langage (impératif) de son choix; bien entendu, plus on s'avance dans le monde de la programmation, plus les particularité du langage de programmation choisi sont prégnantes et "polluent" la conception des "marches à suivre".

- Le choix fait, dans le premier volume, des GNS pour la représentation "graphique" des algorithmes éloigne la phase du "Comment faire faire ," où l'algorithme est conçu et (re)présenté, de celle du "Comment dire ?" où le programme est écrit, dans le respect de la syntaxe d'un langage de programmation.

Mais ces quelques raisons de continuer de dissocier, au début de l'apprentissage, la recherche et la conception d'un algorithme de son expression dans un programme, conduisant donc à deux expressions, côte à côte, de la même chose, ne doit pas faire oublier que, à terme, il est plus efficace d'être capable de concevoir directement la "marche à suivre" en utilisant le langage de programmation. Lorsque le respect des règles syntaxiques du langage de programmation n'est plus un problème et lorsque l'environnement de programmation apporte aide et facilité au concepteur des programmes, il est probablement plus efficace d'amalgamer les étapes du "Comment faire faire ?" et du "Comment dire ?".

Il y a aussi quelques choix plus "techniques" qui sont faits dans ce second volume et qui méritent qu'on s'y attarde. Bien entendu, c'est d'une postface, plutôt que d'un avant-propos, que ces choix devraient faire l'objet d'explication et de justification.

1.4 Les paramètres des procédures

L'essentiel de ce second volume tourne autour de la démarche descendante (rappelez-vous le "top down programming" des années 80). Et l'incarnation dans le langage de programmation de la démarche descendante, ce sont bien entendu les procédures qui la permettent.

Comme on le verra, la manière dont la démarche descendante est amenée et illustrée ici conduit à une utilisation des variables globales et donc à un travail des procédures par "effet de bord". Autrement dit, le rôle d'une procédure est généralement, sur base du contenu de variables préexistantes (= de plus haut niveau) de modifier le contenu d'autres variables préexistantes. Les traces du travail d'une procédure sont donc à chercher au sein de certaines variables que la procédure aura explicitement modifiées.

Dans cette approche, les paramètres ne se justifient évidemment pas, puisque les procédures agissent directement sur des variables de plus haut niveau.

Le chapitre 3 a pour but d'illustrer la nécessité des paramètres lorsqu'on s'écarte de la démarche descendante. Il y a pour moi trois raisons à l'utilité des paramètres :

- Lorsque deux procédures parfaitement jumelles au plan des actions qu'elles accomplissent, sont appelées pour effectuer ces actions sur des variables différentes à chaque appel, il est plus efficace d'écrire une seule procédure, assortie de paramètres qui permettront de préciser, à chaque appel, à quelles variables sont associés les divers paramètres ou encore les valeurs que prennent ces paramètres.
- Lorsque l'appel d'une procédure est effectué par une autre procédure que sa procédure-mère (celle au sein de laquelle elle est logée), elle n'a en général pas directement accès aux variables de cette procédure appelante; dans ce cas, c'est seulement à travers des paramètres (valeurs) que les données nécessaires à son travail doivent être passées à la procédure appelée et au travers d'autres paramètres (variables) qu'elle peut modifier certaines des variables de la procédure appelante (ou accessibles à cette dernière).
- Enfin, le domaine roi de l'usage des paramètres, mais qui n'est pas illustré ici (ce sera pour le troisième volume !), c'est celui de l'approche ascendante (bottom up programming) : on écrit des procédures a priori, sans savoir au sein de quel programme particulier elles viendront prendre place. Dans ce cas, pas question pour ces procédures d'accéder directement à des

variables de plus haut niveau qui n'existent pas encore et dont on ignore la dénomination précise.

En d'autres termes, dans l'approche suivie ici, les procédures ne nécessitent, la plupart du temps, pas de paramètres. Je sais que pour beaucoup de collègues, une procédure est forcément assortie de paramètres, même lorsque ceux-ci ne sont pas nécessaires. Je préfère une approche qui permette de comprendre, même si l'on décide d'utiliser de toute manière des paramètres, quand ils sont nécessaires et quand on pourrait décider de s'en passer.

Je n'ai évidemment pas attendu dix années avant d'apporter une suite au premier volume : cela fait plus de 5 ans que les chapitres 2 et 3 sont écrits. Le chapitre 1, moins urgent pour les enseignements qui m'étaient confiés, a mis un peu plus de temps à être écrit. Ce qui suit est aussi l'occasion de jeter un regard en arrière, sur le premier volume mais également sur la suite, pour juger de la pertinence des choix effectués.

2. Si c'était à refaire

J'ai dit plus haut ce qu'il fallait penser du choix d'une expression des algorithmes qui ne se fasse pas directement dans le langage de programmation. L'usage d'une présentation graphique de type GNS me semble être l'une des raisons essentielles de ce choix. Mais ce type d'expression, utile au tout début, continue-t-il à être utile dans la suite ?

2.1 Le choix des graphes de Nassi-Schneidermann (GNS)

Les raisons qui m'avaient fait retenir ce mode d'expression étaient les suivantes :

- L'autre mode de représentation graphique utilisé dans l'expression des algorithmes, les organigrammes (ou ordinogrammes), qui mettent en avant la structure de branchement, ne conviennent pas pour une approche qui utilise les structures de contrôle de la programmation structurée.
- L'aspect graphique ou schématique des GNS est important pour fixer les concepts de structure de contrôle, essentiels pour qui aborde l'algorithmique et la programmation.
- Le fait que le mode d'expression des GNS soit fort éloigné du Pascal plaide pour l'usage de ceux-ci, si l'on souhaite clairement démarquer l'étape de la conception de la marche à suivre de celle de l'écriture du programme correspondant. Le mode d'expression de type pseudo-code, trop proche du Pascal (c'est du Pascal, en français, sans les begin et les points-virgules), risque de brouiller les étapes et de faire confondre la conception de l'algorithme et l'écriture du programme Pascal correspondant.

Mais si les GNS continuent à constituer un outil intéressant pour le début de l'apprentissage, il faut aussi signaler :

- Les GNS constituent un excellent moyen de présentation des algorithmes, une fois ceux-ci créés, mais la difficulté de tracer ces GNS, et surtout de les modifier, en fait un piètre outil de mise au point des algorithmes à créer.
- La majorité des ouvrages consacrés à l'algorithmique ou à la programmation utilisent les pseudo-codes (aussi appelé langage de description d'algorithme). Les GNS ne sont pratiquement jamais utilisés.

Ce sont ces raisons qui m'ont amené à remplacer graduellement dans le présent ouvrage les GNS par des pseudo-codes.

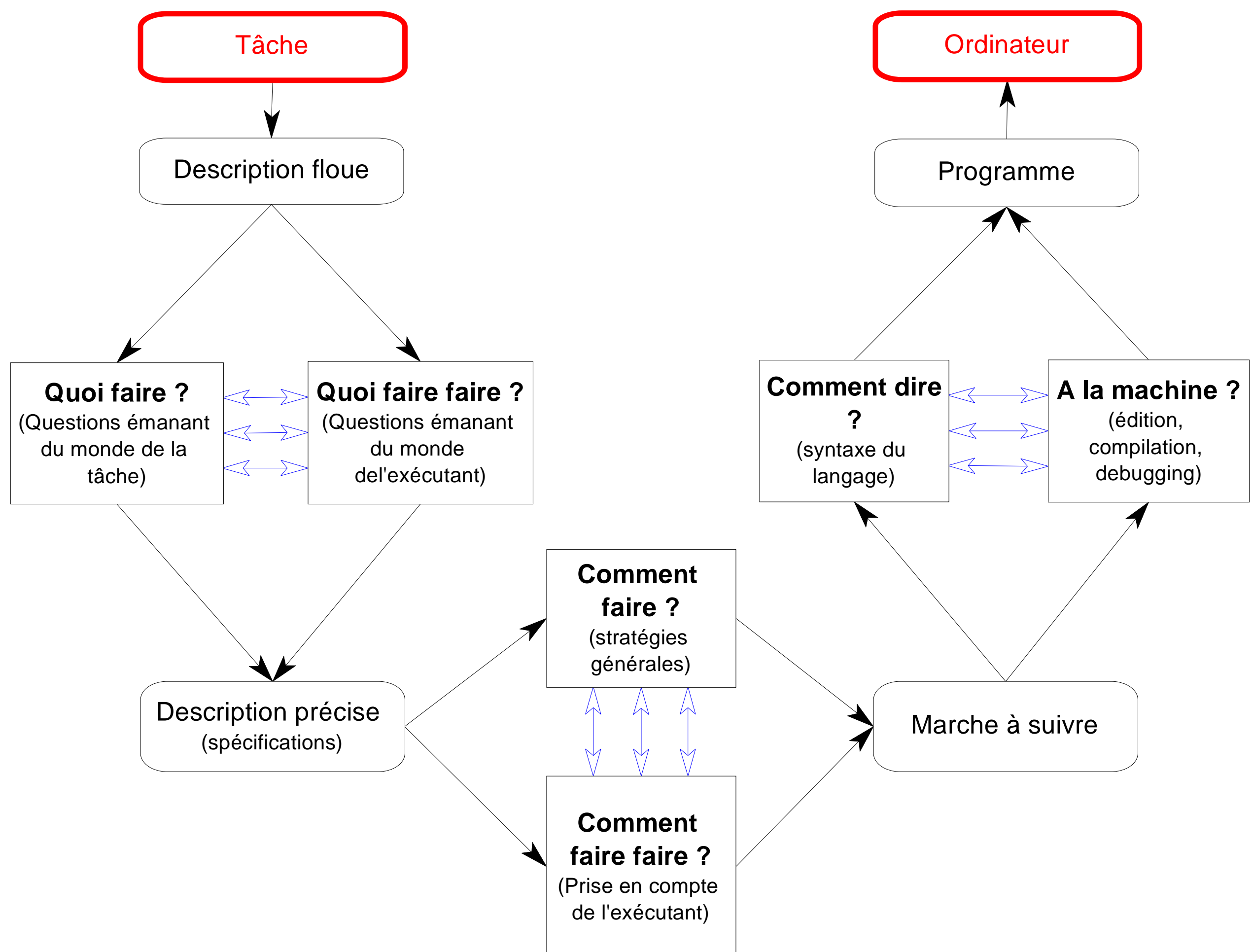
2.2 *La métaphore de l'exécutant-ordinateur*

L'image du "gestionnaire de casiers" (les variables) continue à me sembler tout à fait appropriée. Simplement, je pense que plutôt que de dessiner des casiers "ouverts", dont on voit le contenu, si c'était à refaire, il serait important de les dessiner fermés, repérables seulement par leurs étiquettes.

De cette manière, on ne pourrait plus "d'un simple coup d'oeil" repérer quel est le plus grand des contenus de 3 casiers par exemple; il faudrait préciser quel casier on compare à quel autre, dans quel ordre,...

2.3 *Les étapes du traitement informatique d'une tâche*

Même si l'ensemble des éléments du processus, illustré en page 44 du premier volume, me semble toujours valable, je pense qu'on pourrait présenter les choses d'une manière un peu différente :



Ce schéma rend compte des éléments suivants :

- Dès la première étape, celle où l'on précise complètement quelle est exactement la tâche à traiter, les questions posées viennent de deux préoccupations : d'une part, la tâche elle-même, indépendamment du fait qu'on souhaite en programmer l'exécution, d'autre part, le fait qu'on va devoir faire faire cette tâche par un exécutant dont on connaît les limites. Évidemment, pour le débutant qui commence à découvrir les caractéristiques de ce dernier,

ces questions viennent moins naturellement à l'esprit : mais, petit à petit, cette étape des spécifications mélange un "quoi faire ?" et un "quoi faire faire ?".

Un certain nombre de flèches illustrent le fait que, bien évidemment, ces deux champs de préoccupation sont liés.

- La seconde étape regroupe deux problématiques : celle de la recherche de stratégies générales (en quelque sorte celle des "idées" de traitement à mettre en oeuvre) et celle de la rédaction précise de la marche à suivre, en se limitant strictement, au niveau de l'expression, aux structures de contrôle permises d'une part, aux instructions d'actions élémentaires à destination de l'exécutant d'autre part.
- La troisième étape qui groupe la question de l'expression dans le langage de programmation et le travail "à la machine" acte en quelque sorte le fait que c'est à travers l'environnement de programmation proposé que le texte du programme est élaboré (et non en une version "manuscrite" préalable) : l'écriture du programme se fait à la machine (grâce à un éditeur de texte plus ou moins dédicacé à cette tâche).

3. Quelques questions importantes

La rédaction de cet ouvrage n'aurait peut-être pas été entreprise, étant donné l'effort et le temps nécessaire, si mes enseignements ne m'avaient en quelque sorte amenés à m'en préoccuper. D'une certaine manière, ces textes étaient disponibles, il restait à les présenter correctement et à les grouper.

C'est donc un hasard, si c'est maintenant que ce second volume est rendu disponible. Et, il faut reconnaître qu'on ne peut laisser de côté trois préoccupations essentielles.

3.1 La programmation : développement ou apprentissage

On aura compris que pour qu'il y ait activité de programmation, il faut que la tâche à faire faire soit éloignée des capacités "primitives" de l'exécutant-ordinateur. Dans une perspective d'apprentissage, il faut donc que les briques primitives, mises à disposition de l'apprenti programmeur et avec lesquelles il va devoir construire la marche à suivre qui fera faire la tâche, soient aussi peu nombreuses et aussi élémentaires que possible.

Il y a donc là une différence fondamentale entre les besoins du programmeur professionnel (celui qui développe "vraiment" de nouvelles applications) et ceux de l'apprenti programmeur. Le premier va chercher à réutiliser au maximum de "grosses" briques déjà prêtes (disponibles au sein de librairies de procédures et fonctions, toute faites) afin que l'écriture de son application repose non sur les primitives réelles de l'exécutant, mais sur des morceaux correspondant à des actions plus complexes et qu'il ne reste plus qu'à assembler. L'efficacité du programmeur va donc s'appuyer sur une connaissance aussi large que possible (encyclopédique, en quelque sorte) de procédures et fonctions réutilisables, programmées par d'autres et disponibles.

Il faut placer l'apprenti programmeur dans un tout autre contexte : l'exécutant à programmer doit être aussi simple que possible en terme de primitives disponibles afin que le décorticage de la tâche à programmer oblige à utiliser à bon escient ces modestes primitives et les structures de contrôle organisatrices. Pas question ici de connaissances encyclopédiques de centaines de morceaux tout prêts à l'usage. Chacune des tâches auxquelles le débutant va s'atteler a déjà été programmée des centaines de fois par d'autres (ne serait-ce que par l'enseignant qui assure la formation de ces débutants) : ce n'est évidemment pas une raison pour que l'apprentissage ne repasse pas par ces problèmes, déjà cent fois résolus.

3.2 La programmation : des entrées-sorties ou des traitements

En quinze ans, l'activité de programmation a changé, sinon de nature, du moins de contenu. Auparavant, programmer, c'était passer 10% de son temps à se préoccuper du dialogue avec l'utilisateur (les "entrées-sorties") et 90% à gérer les traitements attendus. Il fut même un temps où l'utilisateur n'existait pas et où, dès lors, aucune gestion de dialogue avec lui n'était à programmer.

Aujourd'hui, dans les environnements fenêtrés, la proportion s'est inversée : construire un programme, c'est se préoccuper à 90% de gérer le dialogue avec l'utilisateur (à travers des boutons, des listes, des boites de dialogue,...), les 10% restants étant consacrés aux traitements.

La programmation sans aucune préoccupation des entrées-sorties, ça devient très vite de l'algorithmique "pure et dure" (et on aura compris que ce n'est pas mon propos); mais passer le plus clair de l'apprentissage à gérer des fenêtres, des boutons et des listes déroulantes, ça n'a en général plus rien à voir avec l'algorithmique du tout. On entre d'ailleurs très souvent alors dans la démarche dénoncée plus haut, celle où programmer demande d'abord de connaître des centaines de primitives, essentiellement dédicacées à la construction de dispositifs de lecture/affichage.

Ce volume, tout en continuant à ne pas ignorer complètement les problèmes (souvent fastidieux et sans intérêt) d'entrées/sorties, continue à mettre l'accent sur les traitements.

3.3 Et l'approche objet ?

Il faut d'abord savoir de quoi l'on parle lorsqu'il est question de programmation objet : s'agit-il de programmer en utilisant des objets graphiques (essentiellement pour la gestion des entrées-sorties) parfois en n'écrivant pas une seule ligne de code, mais simplement en "traînant" les boutons à installer et en garnissant des listes déroulantes toute faites. Dans ce cas évidemment, il ne s'agit ni d'approche objet, ni même de programmation tout court.

Ou bien, il s'agit vraiment d'une approche objet qui utilise pour de bon les concepts et les mécanismes qui la caractérise : encapsulation, héritage, surcharge, polymorphisme. Cette approche est devenue la méthodologie reine des développeurs (à travers des langages comme Java ou C++). La préoccupation de réutilisabilité omniprésente dans cette approche, nous ramène d'ailleurs à une programmation postulant une (très) large connaissance des centaines de classes d'objets déjà disponibles et des milliers de méthodes qu'elles comportent (voir ci-dessus).

Par ailleurs, la philosophie profonde de l'approche objet, est davantage, me semble-t-il, "ascendante" (bottom-up) que descendante ("top down"), même si, comme toute programmation, l'approche est toujours un savant mélange des deux.

De toute manière je pense que :

- L'approche objet s'appuie sur une bonne connaissance des concepts et outils de la programmation impérative (structurée) classique; elle réclame donc des compétences en algorithmique. Ce qui ne veut pas dire qu'il faille forcément une très longue introduction à la programmation procédurale classique avant d'aborder l'approche objet.
- Dans une perspective de premier apprentissage de la programmation à travers l'approche objet, on ne peut faire l'économie d'une profonde réflexion didactique sur la présentation des concepts et sur le balisage du parcours d'apprentissage par des problèmes adaptés et soigneusement choisis. Cette vulgarisation de l'approche objet[1], déjà largement entamée au CeFIS, doit être poursuivie et approfondie.

---

[1] On consultera par exemple :
CHEFFERT J.-L., Comprendre les concepts fondamentaux de la programmation orientée objet. Implémentation

3.4 Et dans l'enseignement secondaire ?

J'ai consacré une partie de ma vie professionnelle aux problèmes du tandem "Informatique - Education". Cette problématique a pris des couleurs différentes suivant les époques. Pendant 10 ans, lorsque l'informatique était une discipline enseignée dans le secondaire, ce fut la didactique de la programmation (qui a débouché sur le premier volume de cet ouvrage et sur l'environnement "Images pour programmer"). Pendant les dix ans qui ont suivi, ce fut davantage la problématique de l'intégration des ordinateurs au sein de l'école et de la détermination des éléments constitutifs d'une "culture de base" pour les utilisateurs des environnements informatisés.

La découverte et l'apprentissage de la programmation ont très largement disparu des programmes des écoles secondaires, en Belgique et ailleurs, en même temps que les cours à option d'informatique étaient supprimés. Je ne reviendrai pas ici sur les causes multiples de cette disparition.

Je reste persuadé qu'à travers des activités correctement planifiées d'algorithmique et de programmation, les élèves peuvent développer des compétences de créativité, de rigueur et d'abstraction. Et cela à travers la résolution de problèmes, ce dernier terme prenant ici un sens extrêmement particulier (la tâche qui sert de point de départ est très facilement compréhensible et peut être aisément cernée, la **faire faire** oblige à énormément de réflexion et de rigueur).

A l'école, l'informatique (discipline enseignée) est censée faire place aux TIC (outils pour aider à l'enseignement et à l'apprentissage). Depuis le temps qu'on en parle, ça finira bien un jour par se faire...

4. Et enfin

Comme en ce qui concerne la première partie, je souhaite que vous preniez du plaisir à parcourir les pages qui suivent. Leur lecture est évidemment plus ardue, les problèmes abordés sont plus complexes et leur traitement est souvent plus long. Il faudrait parfois avoir plusieurs paires d'yeux pour visualiser simultanément des passages différents du texte...

Il ne me reste qu'une dernière demande à vous faire : signalez-moi les erreurs, les passages incompréhensibles ou les explications inappropriées. Dites-moi aussi ce que vous pensez de cet ouvrage. Tout cela est devenu facile : il vous suffit de m'adresser un courrier électronique (à l'adresse charles.duchateau@fundp.ac.be).

Et pour la troisième partie, il vous suffira peut-être de faire oralement vos commentaires en face de votre ordinateur (ou même seulement de les penser) pour qu'ils me parviennent...

Bonne lecture et bon travail.

Charles Duchâteau

A noter : les textes des programmes développés dans cet ouvrage sont disponibles sur disquette (ou sous forme d'un fichier "zippé") et peuvent être obtenus en s'adressant à l'auteur.

---

des concepts en Pascal., Namur : CEFIS, Facultés N-D de la Paix, 1997.
DELACHARLERIE A., Introduction au paradigme "orienté objet", Namur : CEFIS, Facultés N-D de la Paix, 1994.
VANDEPUT E., Programmer avec des objets, Namur : CEFIS, Facultés N-D de la Paix, 2002.

1

Les tableaux

Nous avons observé un exécutant-ordinateur capable de gérer de multiples casiers (variables). Nous allons à présent découvrir qu'il peut également disposer, à côté de ces casiers épars, d'étagères, voire d'armoires…

Nous découvrirons en même temps une forme de répétition particulière : celle dont les boucles successives sont scandées par l'augmentation d'une variable compteur qui démarre à une valeur initiale pour atteindre une valeur terminale, qui marque la fin des répétitions. Traditionnellement, ce type de boucle répétitive est appelée "boucle Pour..." ou "boucle for..." en référence à la manière dont elle s'exprime dans la plupart des langages.

1. Retour sur le calcul de la moyenne

Nous avons traité (au sein des chapitres 4 et 5 du premier volume) le problème de calcul de la moyenne et de la détermination de la plus grande et de la plus petite de 100 données réelles. L'exercice 2.2 (page 180, volume I) proposait d'ailleurs de reprendre la même tâche, mais de chercher des manières de ne plus se limiter à un nombre précis et prédéterminé (100) de données. Nous allons d'abord, pour nous remettre ce problème en tête, évoquer cette question d'une succession de lectures-traitements dont le nombre exact n'est pas connu au moment de l'écriture du programme.

1.1 Énoncé : description floue

Je voudrais que l'utilisateur du programme à concevoir puisse fournir ses données réelles, en nombre quelconque, et obtenir ensuite la moyenne, la plus grande et la plus petite des données ainsi fournies.

> Rappelez-vous : la tâche à programmer se présente le plus souvent d'abord sous la forme d'une description floue qu'il s'agira d'abord de préciser.

1.2 Précisions : quoi faire (faire) ?

Le seul problème c'est que, au moment de rédiger la marche à suivre qui fera lire et traiter la succession des données de l'utilisateur, j'ignore le nombre exact de données qui devront être lues et traitées.

1.2.1 Une première piste sans issue… pour le moment

Si la tâche était à faire par un exécutant-humain face à un utilisateur-humain, voilà à quoi pourraient ressembler leur dialogue :

- (l'exécutant) : "Tu vas me dicter la série des données que je vais traiter. Vas-y!"
- (l'utilisateur) : "14.5"
- (l'exécutant) : "La suivante"
- (l'utilisateur) : "-13"
- (l'exécutant) : "La suivante"
- (l'utilisateur) : "4.5"
- (l'exécutant) : "La suivante"
- (l'utilisateur) : "**C'est tout, je n'ai plus de donnée!**"
- (l'exécutant) : "La moyenne est 2.0; la plus petite donnée est -13.0; la plus grande est 14.5"

Cette manière de procéder n'est pas transposable à ce qui se passera avec l'exécutant-ordinateur. En effet, on peut, en anticipant un peu sur l'écriture de la marche à suivre, prévoir que celle-ci comportera une répétition au cours de laquelle on fera à chaque fois lire une donnée réelle. L'exécutant va donc, à chacune de ces répétitions, trouver une instruction du genre

Lis et place dans *Donnee*,   *Donnee* étant une variable de type **réel**

Tout ira bien lorsque, lors de chacune de ces instructions, l'utilisateur fournira les données réelles attendues : chacune de celle-ci sera saisie par l'exécutant et placée dans le casier *Donnee*, apte à les recevoir. Mais si, à la fin du processus, l'utilisateur fournit comme donnée "**C'est fini!**", il ne s'agit évidemment pas d'une donnée réelle, mais d'une chaîne de caractères. Et lorsque l'exécutant tentera de placer cette dernière dans le casier *Donnee*, qui ne peut accepter que des nombres réels, une erreur se produira immanquablement.

> On notera au passage qu'on pourrait commander des lectures successives non de données réelles, mais de chaînes de caractères. Dans ce cas, il serait simple de faire arrêter le processus lorsque la chaîne "C'est fini" serait fournie. Mais, pour toutes les autres vraies données, qui seraient lues comme des chaînes de caractères, il faudrait avoir un moyen de les transformer en nombres réels. Ainsi, ayant lu, par exemple, la chaîne "14.5", succession des caractères "1", "4", "." et "5", il faudrait pouvoir la transformer en nombre. Nous verrons que c'est évidemment possible.
>
> Dans ce cas, *Donnee* serait évidemment une variable de type chaînes de caractères (string) et il faudrait une autre variable *NombreCorrespondantADonnee*, de type réel, qui contiendrait à chaque fois le nombre correspondant à la chaîne contenue dans *Donnee*.
>
> Il est impératif de bien saisir que les informations de type chaînes de caractères ou nombre sont représentées et traitées fort différemment par l'exécutant-ordinateur, même si nous écrivons la chaîne de caractères "14.5" de la même manière que le nombre 14.5 (voir plus loin, pages 29 et suivantes).

1.2.2 Une solution : la donnée "bidon"

Si on veut concilier le fait que toutes les données fournies par l'utilisateur lors des lectures successives doivent être de type réel (pour prendre place dans la variable réelle *Donnee*), avec la nécessité pour l'utilisateur de pouvoir fournir une donnée marquant la fin, une solution s'offre comme évidente : la donnée signalant la fin doit être de même type réel que toutes les autres (vraies) données, mais s'en distinguer de l'une ou l'autre manière.

 © CeFIS - FUNDP 2002

Une première stratégie serait d'imposer à l'utilisateur la donnée réelle avec laquelle il marquerait la fin des lectures, en lui précisant au tout début du programme, par un avertissement affiché à l'écran, ce que doit être cette donnée de fin.

Il faut évidemment que le programmeur évite d'imposer comme donnée de fin un nombre identique à l'une des vraies données que l'utilisateur fournira. Sauf circonstance particulière, cette imposition d'une donnée de fin est extrêmement problématique. Cela pourrait marcher, par exemple s'il était certain que toutes les données fournies seront positives : on imposerait alors par exemple le nombre -1 à l'utilisateur pour marquer la fin des données.

Dans le cas présent, les spécifications précises n'ont pas encore été décidées, mais il n'y a a priori aucune raison de limiter les données qui pourront être fournies. Impossible donc de décider que -1 ou 0 ou 1000 marqueront la fin des données, puisque ces nombres pourront figurer parmi les vraies données de l'utilisateur.

La solution, dans ce cas, c'est de demander à l'utilisateur lui-même de préciser, avant que les lectures des données ne commencent, quelle donnée il choisit pour marquer la fin. C'est à l'utilisateur qu'incombe donc le choix préalable de la donnée qui marquera la fin. Cette donnée "signal" sera lue et placée dans une variable "signal". Tout au long des lectures, les données lues seront à chaque fois comparées au contenu de la variable "signal' : lorsque la donnée lue sera identique au contenu de la variable signal, les lectures s'arrêteront et cette dernière donnée **ne sera évidemment pas traitée**. Il est bien entendu indispensable que l'utilisateur fournisse comme donnée "signal" une donnée à coup sûr distincte de ses vraies données.

Ce tour de main est fréquemment rencontré en programmation : je le rappellerai habituellement en parlant de donnée **bidon** . Il s'agit donc d'une donnée de même type que toutes les vraies données, mais qui est seulement là pour marquer la fin : elle ne compte pas et ne doit évidemment pas être traitée comme les vraies données. Ici, c'est l'utilisateur qui précisera ce qu'elle est au tout début.

1.2.3 Une autre solution : demander le nombre de données qui seront lues

On pourrait aussi imaginer de faire préciser par l'utilisateur avant le début des lectures, combien de données il y aura à lire. On stockerait ce décompte dans une variable *Nombre* et on pourrait reprendre telle quelle la marche à suivre proposée à la page 177 de la première partie, en changeant seulement le test d'arrêt de la répétition : au lieu de "répéter… jusqu'à ce que *Compteur* = 100", on commanderait "répéter… jusqu'à ce que *Compteur* = *Nombre*".

Mais, il faut admettre que, dans le cas présent, offrir de calculer une moyenne en demandant à l'utilisateur de compter le nombre de données qu'il fournira, c'est lui demander de faire une bonne partie du travail. Il ne resterait plus qu'à lui demander de fournir aussi la somme des données… et le programme deviendrait rigoureusement inutile.

1.2.4 Une dernière manière de procéder

Après chaque lecture de donnée, on pourrait faire explicitement poser la question "On continue oui ou non ?", et faire lire la réponse (dans une variable *Reponse* de type chaîne); la répétition s'effectuerait jusqu'à ce que *Reponse* contienne "NON" (ou toute autre chaîne par laquelle on demanderait à l'utilisateur de marquer la fin).

Mais ceci obligerait l'utilisateur après chaque fourniture de donnée, de fournir aussi la réponse à la question de savoir s'il souhaite poursuivre…

Nous poursuivrons donc l'analyse en utilisant le tour de main de la "donnée bidon".

1.3 Comment faire ?

Les stratégies ont déjà été mises en lumière dans la première partie (pages 174 et suivantes). Simplement, à chaque nouvelle donnée, on vérifiera qu'il ne s'agit pas de la donnée bidon et les lectures-traitements seront repris jusqu'à ce que la donnée lue soit la donnée bidon.

1.4 Comment faire faire ?

La liste des variables est connue (Cf. 1ère partie, page 175) :

- *Donnee*, de type réel, qui contiendra successivement les données lues,
- *Compteur*, de type entier, qui permettra de les compter,
- *Somme*, de type réel, qui accueillera la somme des données,
- *PlusPetite*, de type réel, qui contiendra la plus petite donnée,
- *PlusGrande*, de type réel, qui contiendra la plus grande,

et il faut y ajouter

- *Bidon*, de type réel, qui conservera la donnée bidon choisie au début par l'utilisateur.

1.4.1 Une première tentative…ratée

On peut tenter d'adapter la marche à suivre déjà écrite (1ère partie, page 177) en proposant :

- Affiche 'Donnée bidon ?'
- Lis et place dans *Bidon*
- Affiche 'Donnée ?'
- Lis et place dans *Donnee*
- Place *Donnee* dans *Somme*
- Place *Donnee* dans *PlusPetite*
- Place *Donnee* dans *PlusGrande*
- Place 1 dans *Compteur*
- Répéter :
    - Affiche 'Donnée ?'
    - Lis et place dans *Donnee*
    - Place *Compteur* + 1 dans *Compteur*
    - Place *Somme* + *Donnee* dans *Somme*
    - *Donnee* < *PlusPetite*
        - V : Place *Donnee* dans *PlusPetite*
        - F : *Donnee* > *PlusGrande*
            - V : Place *Donnee* dans *PlusGrande*
            - F :
- jusqu'à ce que *Donnee* = *Bidon* **(1)**
- Affiche *Somme* / *Compteur* **(2)**
- Affiche *PlusPetite*
- Affiche *PlusGrande*

Comme nous avons décidé d'abandonner peu à peu les GNS pour présenter les algorithmes à l'aide d'un langage de description d'algorithme, voici côte à côte les deux constructions correspondantes :

Affiche 'Donnée bidon ?'
Lis et place dans *Bidon*
Affiche 'Donnée ?'
Lis et place dans *Donnee*
Somme ← *Donnee*
PlusPetite ← *Donnee*
PlusGrande ← *Donnee*
Compteur ← *1*
Affiche 'Donnée ?'
Lis et place dans *Donnee*
Compteur ← *Compteur* + 1
Somme ← *Somme* + *Donnee*
Donnee < *Petite*
V
F
PlusPetite ← *Donnee*
Donnee > *PlusGrande*
PlusGrande ← *Donnee*
Donnee = *Bidon* **(1)**
Affiche *Somme* / *Compteur* **(2)**
Affiche *PlusPetite*
Affiche *PlusGrande*

Affiche 'Donnée bidon ?'
Lis et place dans *Bidon*
Affiche 'Donnée ?'
Lis et place dans *Donnée*
Place *Donnee* dans *Somme*
Place *Donnee* dans *PlusPetite*
Place *Donnee* dans *PlusGrande*
Place 1 dans *Compteur*
répeter
    Affiche 'Donnée ?'
    Lis et place dans *Donnee*
    Place *Compteur* + 1 dans *Compteur*
    Place *Somme* + *Donnee* dans *Somme*
    Si *Donnee* < *Petite* alors
        Place *Donnee* dans *PlusPetite*
    sinon
        Si *Donnee* > *PlusGrande* alors
            Place *Donnee* dans *PlusGrande*
jusqu'à ce que *Donnee* = *Bidon* **(1)**
Affiche *Somme* / *Compteur* **(2)**
Affiche *PlusPetite*
Affiche *PlusGrande*

Deux remarques à propos de cette adaptation trop rapide de la version précédente de la marche à suivre à la présente situation :

(1) Le test d'arrêt de la répétition qui consistait à comparer le compteur de données lues à 100 (dans la version originale) a évidemment été modifié : on arrête à présent lorsque la donnée lue est identique à la donnée bidon fixée préalablement (*Donnee* = *Bidon*);

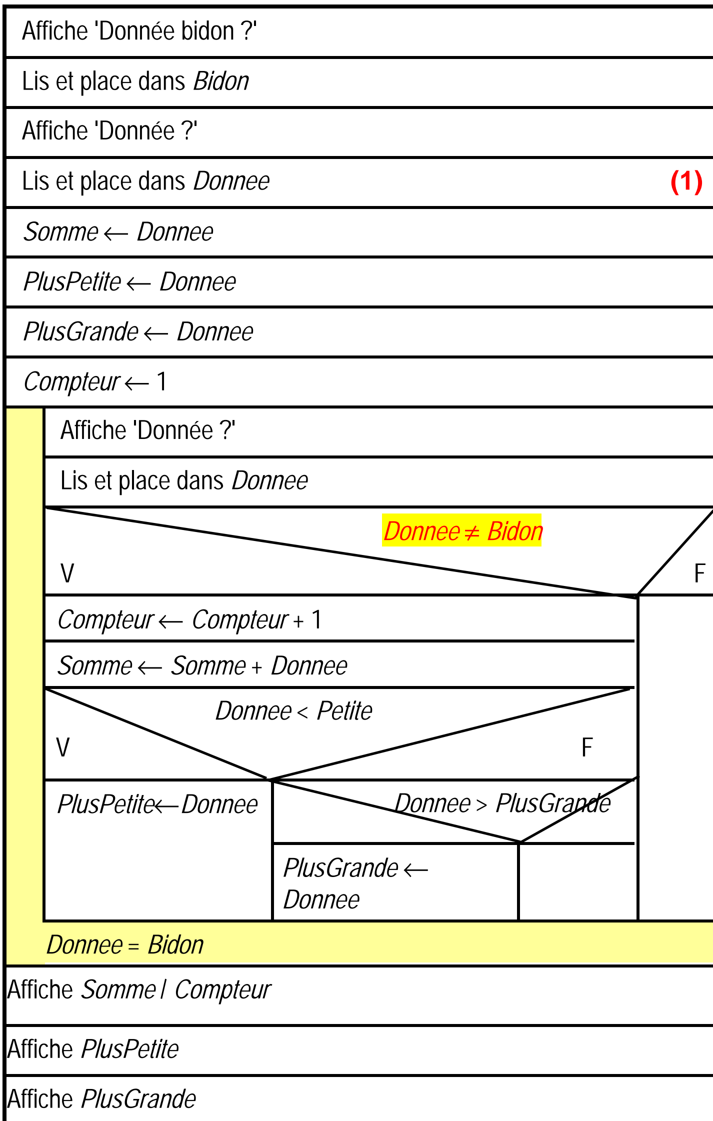
(2) La moyenne n'est évidemment plus *Somme* / 100, mais *Somme* / *Compteur*, *Compteur* contenant le nombre de données lues.

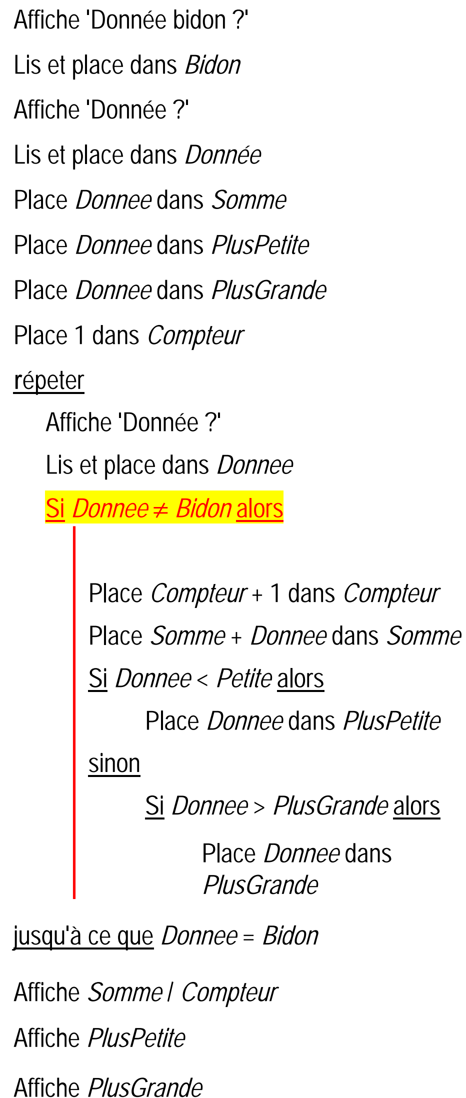
Cette adaptation trop rapide est malheureusement complètement incorrecte : la toute dernière donnée lue, la donnée bidon, qui doit seulement interrompre le processus et ne peut être traitée comme les vraies données, subit ici un traitement absolument identique à toutes les autres.

On vérifiera aisément, par exemple, que si la donnée bidon a été choisie comme 100 (*Bidon* contient donc 100) et si la suite des données fournie est 2, 4, 6, puis 100 pour arrêter, la moyenne calculée et affichée sera 28 (112 / 4) et non 4 comme attendu; de plus la plus grande donnée annoncée sera 100 et non 6. Bref, l'erreur consiste à traiter la donnée bidon comme toute les autres.

1.4.2 Un replâtrage de la première tentative

Si on veut éviter que la donnée bidon terminant la série soit traitée comme les autres, on peut placer un test qui ne permettra le traitement que si la donnée lue n'est pas la donnée bidon comme dans la proposition suivante :





Cette solution est plus satisfaisante; il faut remarquer cependant qu'elle postule qu'il y ait au moins une vraie donnée à traiter. En effet, dans le cas (très improbable) où l'utilisateur fournit comme toute première donnée la donnée bidon (en **(1)**), celle-ci est lue, traitée et n'arrête pas le processus. Il faudra la fournir une seconde fois pour que le processus s'arrête.

Nous voilà une fois de plus au prise avec la difficulté de déterminer si un algorithme est correct. La proposition faite ci-dessus est-elle correcte? En programmation, le terme "correct" n'a pas un sens absolu, mais signifie seulement "conforme aux spécifications" : la marche à suivre imaginée provoque-t-elle chez l'exécutant-ordinateur un comportement conforme à ce qui avait été demandé et précisé dans la description préalable (le "Quoi faire ?").

Nous n'avons rien dit ci-dessus à propos de ce qui est attendu lorsque l'utilisateur, après avoir précisé quelle est la donnée bidon qu'il a choisie, fournit comme première (et seule) donnée cette donnée bidon. Le programmeur pourra bien entendu se retrancher derrière la difficulté de prévoir un comportement aussi étonnant et aberrant de l'utilisateur. Mais, si vous saviez combien ces comportements sont monnaie courante…

La proposition ci-dessus est donc "correcte" pour autant que l'utilisateur fournisse au moins une vraie donnée, mais insatisfaisante si l'utilisateur ne fournit comme seule donnée que la donnée bidon (qu'il venait de préciser).

1.4.3 Une dernière proposition mieux construite

Voici une dernière solution qui fait usage d'une boucle Tant que…. Elle a l'avantage de résister à un comportement de l'utilisateur qui ne fournit comme seule donnée que la donnée bidon.

Affiche 'Donnée bidon ?'
Lis et place dans *Bidon*
Compteur ← *0* **(3)**
Somme ← 0 **(4)**
Affiche 'Donnée ?'
Lis et place dans *Donnee*
PlusPetite ← *Donnee* **(1)**
PlusGrande ← *Donnee* **(2)**
Donnee ≠ *Bidon* **(5)**
Compteur ← *Compteur* + 1 **(6)**
Somme ← *Somme* + *Donnee* **(7)**
Donnee < *Petite* **(11)**
V F
PlusPetite ← *Donnee*
Donnee > *PlusGrande* **(12)**
PlusGrande ← *Donnee*
Affiche 'Donnée ?' **(8)**
Lis et place dans *Donnee* **(9)**
Compteur ≠ 0 **(10)**
V F
Affiche *Somme* / *Compteur* **(13)**
Affiche 'Pas de donnée!!!'
Affiche *PlusPetite*
Affiche *PlusGrande*

Affiche 'Donnée bidon ?'
Lis et place dans *Bidon*
Place 0 dans *Compteur* **(3)**
Place 0 dans *Somme* **(4)**
Affiche 'Donnée ?'
Lis et place dans *Donnée*
Place *Donnee* dans *PlusPetite* **(1)**
Place *Donnee* dans *PlusGrande* **(2)**
Tant que Donnee ≠ Bidon **(5)**
    Place *Compteur* + 1 dans *Compteur* **(6)**
    Place *Somme* + *Donnee* dans *Somme* **(7)**
    Si *Donnee* < *Petite* alors **(11)**
        Place *Donnee* dans *PlusPetite*
    sinon
        Si *Donnee* > *PlusGrande* alors **(12)**
            Place *Donnee* dans *PlusGrande*
    Affiche 'Donnée ?' **(8)**
    Lis et place dans *Donnee* **(9)**
Si *Compteur* ≠ 0 alors **(10)**
    Affiche *Somme* / *Compteur* **(13)**
    Affiche *PlusPetite*
    Affiche *PlusGrande*
Sinon
    Affiche 'Pas de donnée!!!'

Quelques commentaires sont indispensables :

- on commence par faire lire la donnée bidon puis la première donnée;
- on applique ensuite un traitement spécifique à la toute première donnée, **sachant qu'elle recevra ensuite (dans le corps de la boucle Tant que, en (6), (7) et (11)) le même traitement que toutes les données**; c'est ceci qui explique que :
    - en **(3)**, on place le *Compteur* à 0, puisque celui-ci sera incrémenté en **(6),**
    - en **(4)**, on met la *Somme* à 0, puisque celle-ci se verra ajouter la *Donnee* en **(7);**
- après ce traitement spécifique à la toute première donnée on entre dans la boucle Tant que… Il faut remarquer que cette boucle ne commencera que si *Donnee* ≠ *Bidon*; si donc la toute première donnée fournie par l'utilisateur est d'emblée la donnée bidon, le corps de la boucle Tant que… n'est pas exécuté du tout (et on se retrouve en **(10)**);

- on remarquera qu'en ce qui concerne la toute première donnée lue (hors de la boucle Tant que), les tests "*Donnee < PlusPetite*" **(11)** et "*Donnee > PlusGrande*" **(12)** seront toujours négatifs puisque on a exigé (en **(1)** et **(2)**) que *PlusPetite* et *Plusgrande* soient égales à *Donnee;*
- il faut bien saisir que le corps de la boucle Tant que comporte successivement d'abord le traitement de la donnée précédemment lue, ensuite la lecture de la donnée suivante (en **(9)**); puis ça recommence (traitement - lecture d'une nouvelle donnée) pour autant que la donnée lue ne soit pas la donnée bidon;
- il faut noter qu'après la lecture de la dernière donnée (en **(9)**), qui est forcément la donnée bidon, le test (en **(5)**) fait que la boucle s'interrompt : la dernière donnée (bidon) n'est donc pas traitée et on se retrouve directement en **(10)**;
- le test "*Compteur* $\neq$ 0" (en **(10)**) est indispensable pour ne pas provoquer (en **(13)**) une erreur due à une division par 0, dans le cas où aucune donnée n'a été lue; en effet, dans le cas où l'utilisateur ne fournit comme première et seule donnée que la donnée bidon, on n'entre pas dans la boucle Tant que… et le *Compteur* reste égal à 0, comme demandé en **(3)**.

1.4.4 Que retenir ?

On peut noter que dans tous les cas où l'on effectue des lectures-traitements de données clôturées par une donnée bidon, la structure de la marche à suivre est généralement du type suivant :

Lecture de la première donnée	
Traitement spécifique à la première donnée (sachant qu'elle recevra ensuite le traitement commun à toutes les données)	
Tant que la donnée n'est pas bidon	
	Traitement de la donnée (commun à toutes les données)
	Lecture d'une nouvelle donnée
Suite	

Lecture de la première donnée
Traitement spécifique à la première donnée (sachant qu'elle recevra ensuite le traitement commun à toutes les données)
Tant que la donnée n'est pas bidon
    Traitement de la donnée (commun)
    Lecture d'une nouvelle donnée
Suite

1.5 Comment dire ?

Nous voici en mesure de traduire cette dernière proposition en un programme Pascal, en débutant par la version (presque) "nue" du programme :

```
program MOYENNE;
(* programme "nu" *)

uses WinCRT;                                (1)
var Donnee,
    Somme,
    Bidon,
    PlusPetite,
    PlusGrande
    : real;
    Compteur : integer;

begin
write('Donnée bidon : ');
readln(Bidon);
Compteur:=0;
Somme:=0;                                   (4)
write('Donnée : ');
readln(Donnee);
PlusPetite:= Donnee;
PlusGrande:=Donnee;
while Donnee <> Bidon do
```

```
  begin
  Compteur:=Compteur + 1; (* ou Compteur :=succ(Compteur) *)
  Somme:=Somme+Donnee;
  if Donnee < PlusPetite then
    PlusPetite:=Donnee
  else
    if Donnee > PlusGrande then
      PlusGrande:=Donnee;
  write('Donnée : ');
  readln(Donnee);
  end;
if Compteur<>0 then
  begin
  writeln('La moyenne est  ',Somme/Compteur:12:3);          (2)
  writeln('La plus plus petite donnée est',PlusPetite:12:3);
  writeln('et la plus grande donnée est ',PlusGrande:12:3);
  end
else
  writeln('Vous ne m''avez pas fourni de données !')        (3)
end.
```

Quelques commentaires :

(1) la mention "uses WinCrt" est indispensable au début des programmes Pascal écrit dans l'implémentation Turbo Pascal sous Windows; sans elle certaines possibilités d'affichage (et autres) ne sont pas possible. Dans les versions antérieures (sous MS-DOS), la mention correspondante était "uses CRT";

En réalité la mention "uses …" est l'appel à ce que l'on nomme une librairie contenant un certain nombre de procédures (et de fonctions) qui viennent en quelque sorte enrichir la base du langage Pascal lui-même. "Uses WinCRT" permet que dans la suite, on fasse appel à des procédures comme ClrScr (pour effacer l'écran) ou GoToXY (pour envoyer le curseur d'affichage à un endroit précis de l'écran).

En bref, la mention "uses WinCrt" est obligatoire, juste après l'en-tête "program…".

(2) Lors de l'affichage d'une quantité de type réel (comme *Somme/Compteur* ou *PlusGrande*), il est possible d'exiger que les données réelles affichées comportent un certain nombre de caractères au total ainsi qu'un certain nombre de chiffres après la virgule ou plutôt le point décimal).

En mentionnant *Somme/Compteur*:**12:3**, on demande que le réel ainsi affiché comporte douze caractères au total et trois chiffres après la virgule. Voici ce que donne cet affichage :

```
Donnée bidon : -1
Donnée : 56.4
Donnée : 23
Donnée : 35.7
Donnée : 13.5
Donnée : -1
La moyenne est        32.150
La plus plus petite donnée est      13.500
et la plus grande donnée est      56.400
```

On notera qu'il y a bien ajout de 0 à la fin de la partie décimale, pour obtenir 3 chiffres en tout et que les 12 caractères demandés au total font apparaître de nombreux blancs avant les chiffres des nombres affichés.

Voici ce que donne la demande suivante :

```
writeln('La moyenne est  ',Somme/Compteur:6:1);
writeln('La plus plus petite donnée est',PlusPetite:6:1);
```

```
writeln('et la plus grande donnée est ',PlusGrande:6:1);
```

```
Donnée bidon : -1
Donnée : 56.4
Donnée : 23
Donnée : 35.7
Donnée : 13.5
Donnée : -1
La moyenne est    32.2
La plus plus petite donnée est  13.5
et la plus grande donnée est   56.4
```

On notera que comme un seul chiffre décimal est exigé, il y a arrondi de cette partie décimale : 32.15 est donc affiché comme 32.2.

Enfin, si ces précisions au niveau de l'affichage ne sont pas données comme dans :

```
writeln('La moyenne est  ',Somme/Compteur);
writeln('La plus plus petite donnée est',PlusPetite);
writeln('et la plus grande donnée est ',PlusGrande);
```

on obtient

```
Donnée bidon : -1
Donnée : 56.4
Donnée : 23
Donnée : 35.7
Donnée : 13.5
Donnée : -1
La moyenne est   3.2150000000E+01
La plus plus petite donnée est 1.3500000000E+01
et la plus grande donnée est  5.6400000000E+01
```

l'écriture E+01 signifiant "x 10^1". Dès lors 3.2150000000E+01 est 3.215 x 10^1 soit 32.15.

Faut-il redire une fois de plus que cette "cuisine" technique à propos de détails propres au langage ou même à son implémentation, s'ils sont utiles pour finir par écrire des programmes fournissant des résultats agréables à lire, n'ont rien à voir avec le cœur de l'activité de programmation, entendue comme la conception d'algorithmes.

(3) On notera la répétition de l'apostrophe au sein d'une constante de type chaîne de caractères.

(4) On notera que lorsque des données réelles sont attendues, que ce soit de la part de l'utilisateur ou de la part du programmeur (dans l'écriture du programme), on peut écrire des données entières. Ainsi, la variable *Somme* est réelle et on a écrit

```
Somme := 0
```

plutôt que

```
Somme := 0.0
```

et dans les exemples d'exécution du programme donnés ci-dessus, l'utilisateur a fourni 23 ou -1 (qui sont des entiers) alors que la variable qui doit les accueillir est de type réel.

Et voici la version "habillée" du programme, en tenant compte de l'utilisateur et du lecteur :

```
program MOYENNE;
uses WinCRT;
(* Il fait calculer la moyenne, la plus petite et la plus grande d'une série
   de données (terminée par une donnée bidon préalablement choisie par l'utilisateur) *)

var Donnee,
      (* qui contiendra chaque donnée à traiter *)
```

```
    Somme,
    (* qui contiendra la somme des données *)
    Bidon,
    (* qui contiendra la donnée bidon, celle qui provoquera dans la suite l'arrêt des lectures *)
    PlusPetite,
    (* qui contiendra la plus petite des données *)
    PlusGrande
    (* qui contiendra la plus grande *)
    : real;
    Compteur : integer;
    (* pour compter les données lues *)

begin
clrscr;
writeln('Vous allez me fournir une série de données réelles et je vous');
writeln('en fournirai la moyenne, la plus petite et la plus grande.');
writeln('Avant de commencer à fournir les données, donnez-moi d''abord celle');
writeln('avec laquelle vous terminerez la liste tantôt : elle ne sera pas ');
writeln('considérée comme une vraie donnée, mais servira seulement à arrêter');
write('Quelle est cette donnée pour l''arrêt : ');
readln(Bidon);
writeln;writeln; (* pour passer deux lignes *)
(* initialisations *)
Compteur:=0;
Somme:=0;
(* lecture de la première donnée *)
write('Donnée : ');
readln(Donnee);
(* traitement spécifique de la première donnée *)
PlusPetite:= Donnee; PlusGrande:=Donnee;
(* traitements et lectures successives *)
while Donnee <> Bidon do
  begin
  Compteur:=succ(Compteur);
  Somme:=Somme+Donnee;
  if Donnee < PlusPetite then
    PlusPetite:=Donnee
  else
    if Donnee > PlusGrande then
      PlusGrande:=Donnee;
  write('Donnée : ');
  readln(Donnee);
  end;
if Compteur<>0 then
  begin
  writeln('La moyenne est  ',Somme/Compteur:12:3);
  writeln('La plus plus petite donnée est',PlusPetite:12:3);
  writeln('et la plus grande donnée est ',PlusGrande:12:3);
  end
else
  writeln('Vous ne m''avez pas fourni de données !');
end.
```

2. Et toujours la moyenne…

Nous allons modifier très légèrement l'énoncé précédent pour déboucher sur… un problème **insoluble** dans l'état actuel de ce que nous connaissons

2.1 Enoncé

On souhaite toujours faire lire la moyenne d'une série de données; mais cette fois, on souhaite à la fin des lectures un affichage qui donne la moyenne des données, puis, pour chacune des données lues, son numéro dans la série, sa valeur et le pourcentage qu'elle représente par rapport à la moyenne des données. On notera qu'on ne demande plus l'affichage de la plus petite et la plus grande des données lues.

2.1.1 Quoi faire ?

Pour préciser la forme que devront prendre les résultats affichés par le programme à écrire, nous présenterons dorénavant, fort souvent, des copies d'écrans reprenant les affichages attendus.

Ici, après un avertissement de l'utilisateur et la lecture de la donnée bidon :

```
Vous allez me fournir une série de données réelles et je vous
en fournirai la moyenne, puis, pour chaque donnée, le pourcentage
représenté par rapport à cette moyenne.
Attention! Vous avez droit à 100 données au plus.
Donnez-moi d'abord la donnée avec laquelle vous terminerez
tantôt la liste : 0
```

on procèdera à la lecture de la série des données

```
Vous allez me fournir une série de données réelles et je vous
en fournirai la moyenne, puis, pour chaque donnée, le pourcentage
représenté par rapport à cette moyenne.
Attention! Vous avez droit à 100 données au plus.
Donnez-moi d'abord la donnée avec laquelle vous terminerez
tantôt la liste : 0


Donnée : 56
Donnée suivante : 32
Donnée suivante : 89
Donnée suivante : 23.7
Donnée suivante : 35
Donnée suivante : 0_
```

et une fois la série terminée, les résultats seront affichés sous la forme suivante :

```
La moyenne est de       47.140
Donnée 1 :   56.0; Pourcentage : 119
Donnée 2 :   32.0; Pourcentage :  68
Donnée 3 :   89.0; Pourcentage : 189
Donnée 4 :   23.7; Pourcentage :  50
Donnée 5 :   35.0; Pourcentage :  74
_
```

Comme on le voit :

- on limite à 100 au plus le nombre de données que l'utilisateur pourra fournir (on comprendra pourquoi dans la suite); mais on ne demande pas que le programme à écrire réagisse d'une manière particulière si plus de 100 données sont fournies;
- c'est à nouveau une donnée bidon qui permettra de marquer la fin des données;
- l'affichage de la moyenne se fera avec 3 chiffres décimaux; chaque donnée sera ensuite rappelée avec un numéro d'ordre, sa valeur (arrondie, avec un seul chiffre décimal) et le pourcentage que la donnée représente par rapport à la moyenne calculée (arrondi, sans chiffre décimal)

 © CeFIS - FUNDP 2002

2.2 *Comment faire retenir l'ensemble des données lues*

On est bien entendu confronté ici à un problème particulier : pour pouvoir calculer la moyenne des données, il faut avoir acquis toutes ces données, mais on ne peut plus se permettre de les oublier au fur et à mesure puisque, une fois la moyenne calculée, il faut lui comparer chaque donnée pour connaître le pourcentage de cette moyenne qu'elle représente.

> On pourrait imaginer une solution désespérée : demander à l'utilisateur de fournir à deux reprises l'ensemble des données : la première fois, elles serviraient à calculer la moyenne, la seconde fois, à préciser le pourcentage représenté par rapport à la moyenne. Outre le fait que les affichages ne seraient pas exactement ceux demandés dans le "Quoi faire ?", cette solution est évidemment insensée.

2.2.1 Une solution impraticable ; des variables distinctes pour retenir les données

On le devine, il faut ici que toutes les données soient "retenues" par l'exécutant et qu'elles prennent donc place côte à côte dans des casiers prévus pour les accueillir. Ce que nous connaissons jusqu'ici des possibilités de l'exécutant, nous entraîne à prévoir des casiers différents, chacun avec son nom propre. On pourrait avoir :

A	*B*	*C*	*D*	*E*	*F*	*G*	...
							...

Ou en choisissant d'autres noms pour les variables :

A1	*A2*	*A3*	*A4*	*A5*	*A6*	*A7*	...
							...

Mais cette solution conduirait à une déclaration des variables, absolument impraticable, du genre :

```
var A, B, C, D, E, F, G, .... : real;
```

ou encore

```
var A1, A2, A3, A4, A5, A6, A7, .... : real;
```

sachant que bien évidemment la liste complète devrait en être dressée et qu'**on ne peut se contenter de finir avec des points de suspension** ... .

Même en limitant à 100 au maximum les données à acquérir pour en garnir ces casiers, il faut reconnaître que cette déclaration est longue et fastidieuse. Une fois ces casiers déclarés, il faudra les remplir, par lecture, des données à y placer. Malheureusement, et c'est là le vrai nœud du problème, on ne peut pas écrire une boucle répétitive (comme on l'avait fait lors du calcul simple de moyenne). Ce serait plutôt quelque chose du type d'une longue suite de lectures :

```
readln(A);
readln(B);
readln(C);
readln(D);
readln(E);
readln(F);
readln(G);
readln(H);
... ...
```

mais **sans point de suspension**...

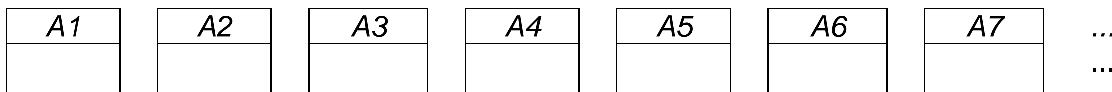
De plus comment pourrait on combiner cette longue suite d'instructions de lecture avec un arrêt commandé par une donnée bidon ? Et, au moment de l'affichage des résultats, il faudrait recommencer une longue série d'instructions writeln...

Le vrai problème avec des variables distinctes comme celles utilisées ici, et même si nous les baptisons *A1, A2, A3,...*, c'est que les symboles 1, 2, 3 intervenant dans l'écriture des noms de variables n'ont rien à voir avec les entiers 1, 2, 3,... manipulés par ailleurs par l'exécutant. *A1, A2, A3* sont des noms de casiers qui n'ont pas plus à voir avec les entiers 1, 2 et 3 que *A*, *B*, et *C*.

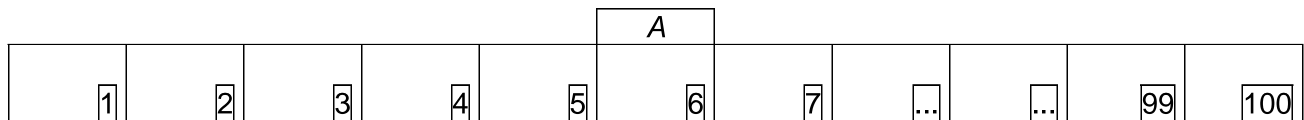
Il faudrait pourtant que le remplissage de ces casiers et le travail qui les mette successivement en oeuvre puisse se faire au sein de boucles répétitives. C'est ce que va permettre le concept développé ici, celui de **tableau**.

2.2.2 Étagère et cie

En réalité, à travers pratiquement tous les langages de programmation, l'exécutant-ordinateur peut gérer non seulement des variables distinctes comme

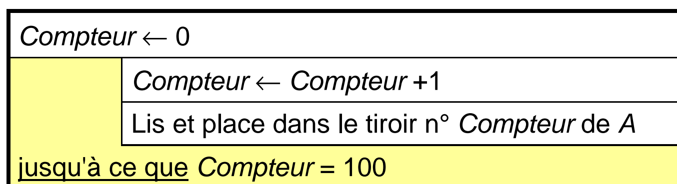


mais aussi des "étagères" comme :

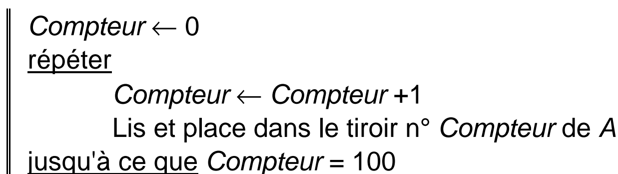


Quelle sont donc les différences entre les deux ?

- D'abord, là où il y avait 100 noms de variables (*A1, A2, A3, ... A100*), il n'y a plus qu'un seul nom d'étagère (*A*).
- Cette étagère comporte cependant une centaine de compartiments (nous dirons pour un moment, de "tiroirs") côte à côte; chacun de ces tiroirs est repéré par une étiquette (1, 2, 3, ... 100), ces étiquettes de tiroir étant des **nombres entiers**, comme ceux que manipule par ailleurs l'exécutant.
- Nous allons donc pouvoir désigner un tiroir de cette étagère en précisant qu'il s'agit de l'étagère *A* et que le tiroir souhaité porte tel numéro (**entier**); nous pourrons même, au moment de préciser l'entier qui numérote un tiroir, écrire "c'est le tiroir *Compteur*", *Compteur* étant un casier entier. On pourra donc très aisément faire remplir par lecture les 100 tiroirs de l'étagère avec une marche à suivre comme :



ou sous une autre forme :



On pourrait bien entendu de la même manière, faire imprimer par exemple les tiroirs 25 à 75 de la même étagère :

Compteur ← 24	
	Compteur ← *Compteur* +1
	Affiche le tiroir n° *Compteur* de *A*
jusqu'à ce que *Compteur* = 75	

ou encore

```
Compteur ← 24
répéter
        Compteur ← Compteur +1
        Affiche le tiroir n° Compteur de A
jusqu'à ce que Compteur = 75
```

Nous disposons donc de l'outil qu'il nous faudra pour emmagasiner côte à côte les données lues : il nous suffira de réserver une étagère à 100 tiroirs.

Bien entendu, presque tous les détails restent à régler : comment (en Pascal) déclare-t-on une étagère ? Comment en désigne-t-on tel tiroir ? Que peuvent être les étiquettes des tiroirs ?

Précisons simplement, avant de poursuivre, que la définition d'une variable-étagère *Donnees* destinée à contenir les données réelles qui seront lues, s'écrira en Pascal :

```
var Donnees : array[1..100] of real
```

Nous aurons ainsi réservé une étagère, nommée *Donnees*, comportant 100 tiroirs, étiquetés de 1 à 100 et susceptibles de contenir des réels.

2.3 Retour au problème : comment faire ?

La stratégie globale est immédiate

- On se renseigne sur la donnée bidon
- On va lire des données successivement et tant que ce n'est pas la donnée bidon
  - on compte la donnée
  - on l'ajoute à la somme
  - on la place dans le tiroir adéquat de l'étagère *Donnees*
- Dans une deuxième phase
  - on calcule et affiche la moyenne
  - on passe en revue tous les tiroirs pertinents de *Donnees* et on affiche leur contenu, divisé par la moyenne calculée et multiplié par 100

2.4 Comment faire faire ?

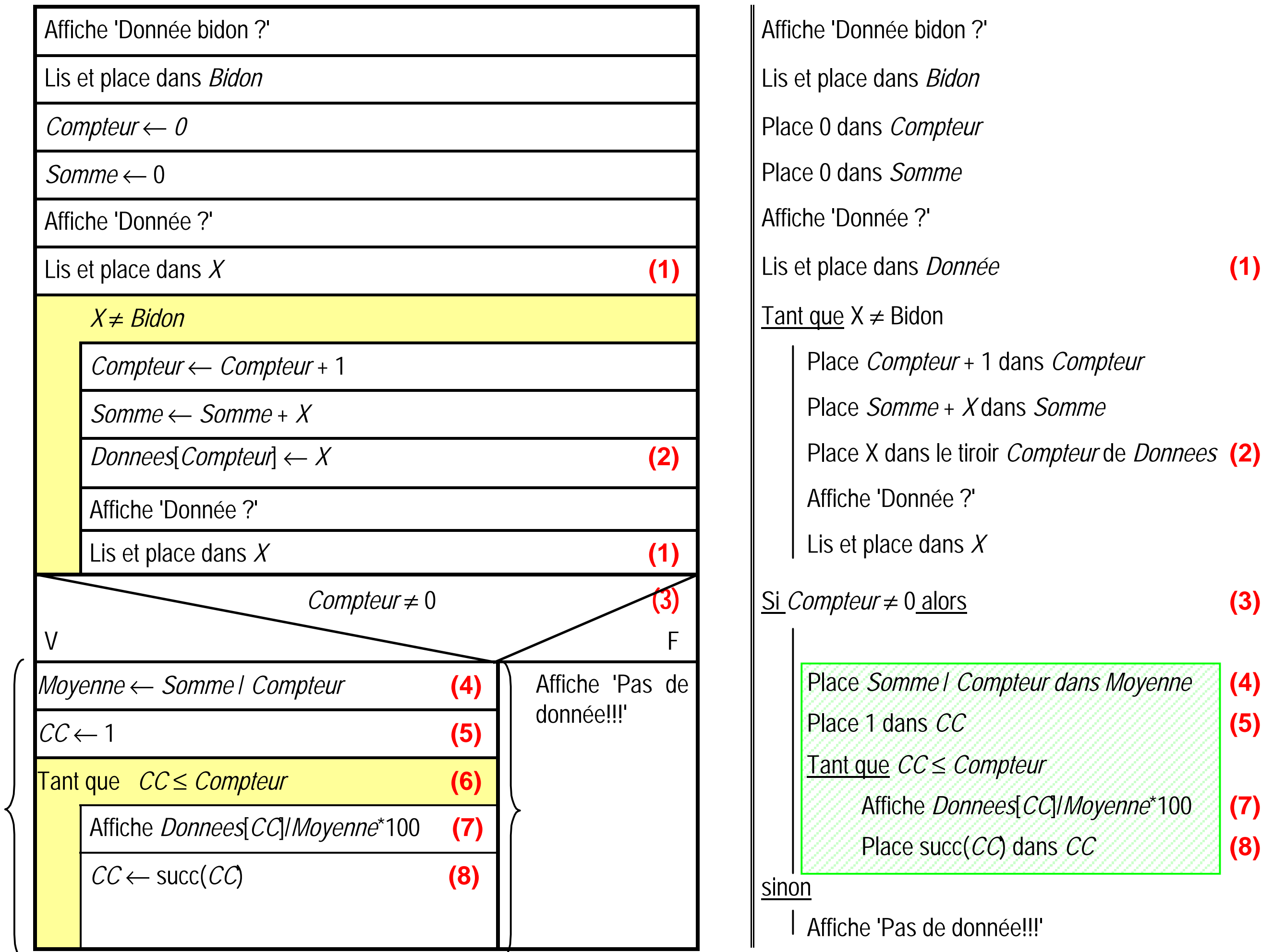
2.4.1 Une première proposition.

Nous pouvons dresser la liste des variables qui sont nécessaires :

- *Donnees*, étagère à 100 tiroirs réels, étiquetés de 1 à 100,
- *Bidon*, de type réel, qui contiendra la donnée bidon précisée par l'utilisateur,
- *X*, de type réel, qui contiendra chaque donnée lue, avant son transfert dans le tiroir adéquat de *Donnees*,
- *Somme*, de type réel, où s'accumulera peu à peu la somme des données lues,

- *Moyenne*, de type réel, qui contiendra la moyenne,
- *Compteur*, de type entier, pour compter les données lues,
- *CC*, de type entier, compteur qui servira au parcours de l'étagère *Donnees* pour calculer les pourcentages par rapport à la moyenne et les afficher.

Et l'algorithme correspondant, sous ses deux formes :



Quelques commentaires :

(1) On notera que lors des lectures successives des données, celles-ci sont déposées dans la variable *X* et pas directement dans le tiroir adéquat de l'étagère *Donnees*; on pourra vérifier qu'il est plus compliqué d'écrire la marche à suivre lorsqu'on fait placer la donnée lue directement dans le tiroir adéquat de *Donnees* (ce qui est pourtant permis).

(2) Pour désigner le tiroir n° *Compteur* de l'étagère *Donnees*, on écrira en Pascal :

```
Donnees[Compteur]
```

C'est cette écriture que nous avons déjà utilisée ici.

(3) A nouveau, nous avons placé un test qui permet d'éviter une erreur dans le cas (improbable) où l'utilisateur ne fournirait comme seule donnée que la donnée bidon.

(4) On utilise une variable *Moyenne* pour accueillir la moyenne des données lues; c'est le contenu de cette variable qui servira au calcul des pourcentages et qui sera affiché.

(5) Ce sont les tiroirs ayant les numéros entre 1 et *Compteur* de l'étagère *Donnees* qui contiennent les données lues; pour parcourir ces tiroirs, on a besoin d'une variable qui

prendra successivement les valeurs allant de 1 à *Compteur*, au sein d'une boucle. On a choisi ici une boucle Tant que... (on verra pourquoi dans la suite). Il aurait probablement été plus naturel de choisir une boucle Répéter... sous la forme :

$CC \leftarrow 0$
$CC \leftarrow$ succ(CC)
Affiche *Donnees*[*CC*]/*Moyenne**100
CC = *Compteur*

$CC \leftarrow 0$
Répeter
    $CC \leftarrow$ succ(CC)
    Affiche *Donnees*[*CC*]/*Moyenne**100
jusqu'à ce que *CC* = *Compteur*

(6) La boucle permettant les affichages successifs se poursuit donc tant que *CC* reste inférieur ou égal à *Compteur* (dernier numéro de tiroir à visiter); dès que cette valeur est dépassée, la boucle s'arrête.

(7) Si on se réfère à la description des affichages exigés, l'instruction d'affichage sera un peu plus complexe que ce qui est rappelé dans la marche à suivre. En réalité, et en anticipant sur l'écriture du programme en Pascal, on aura :

```
writeln( 'Donnée ',CC,' : ',
         Donnees[CC]:6:1,
         '; Pourcentage : ',
         Donnees[CC]/Moyenne*100:3:0);
```

Ainsi, au moment où par exemple *CC* vaut 4, si le contenu du tiroir 4 est 135.67 et si la moyenne vaut 121.33, on aura (en rendant visibles les espaces sous la forme _) :

```
writeln(
'Donnée_',CC,'_:_',Donnees[CC]:6:1,';_Pourcentage_:_',Donnees[CC]/Moyenne*100:3:0);
```

```
Donnée_4_:__135.7;_Pourcentage_:_112
```

(8) Pour passer à la valeur suivante du compteur *CC*, on écrit

$$CC \leftarrow \text{succ}\ (CC);$$

on aurait bien entendu pu écrire

$$CC \leftarrow CC + 1;$$

Rappelons qu'en ce qui concerne les données de type entier ou caractère, la fonction succ fournit le suivant et la fonction pred le prédécesseur (voir page 39).

On notera encore les deux éléments suivants :

- En conformité avec les spécifications, rien n'est prévu dans ce programme pour empêcher que plus de 100 données puissent être fournies par l'utilisateur; si tel était le cas, il y aurait des problèmes en **(2)** puisque *Compteur* deviendrait supérieur à 100, et face à

$$Donnees[Compteur] \leftarrow X$$

on tenterait d'accéder à un tiroir de *Donnees* inexistant (puisque les étiquettes de tiroirs s'arrêtent à 100).

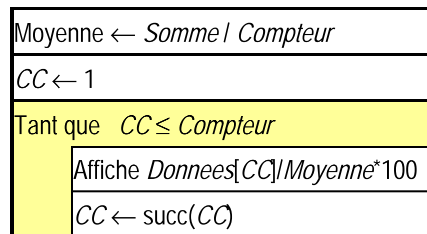
- L'algorithme décrit semble satisfaisant. Et pourtant, il reste un cas "pathologique" qui risque d'entraîner des problèmes.

? Tentez donc de prévoir ce qui se passera si vous fournissez comme (vraies) données : 17.6, -14.8, 31.4, -21.1 et -13.1.

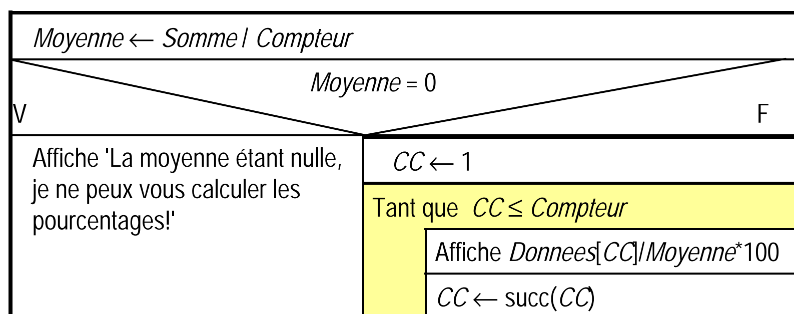
On constate immédiatement que la moyenne de ces données est 0. On aura donc des problèmes en **(7)** lors de la division de chaque donnée par cette moyenne.
Bien évidemment, ceci ne se produira pas, par exemple, si toutes les données sont positives (ou toutes négatives). On peut d'ailleurs se demander quel sens a le pourcentage par rapport à la moyenne lorsqu'on est face à un mélange de données positives ou négatives.
Mais, quoi qu'il en soit, rien dans les spécifications n'empêchait un mélange de données positives et négatives. L'algorithme fourni n'est donc pas correct et il faut bien entendu en retravailler la fin.

2.4.2 Corrections

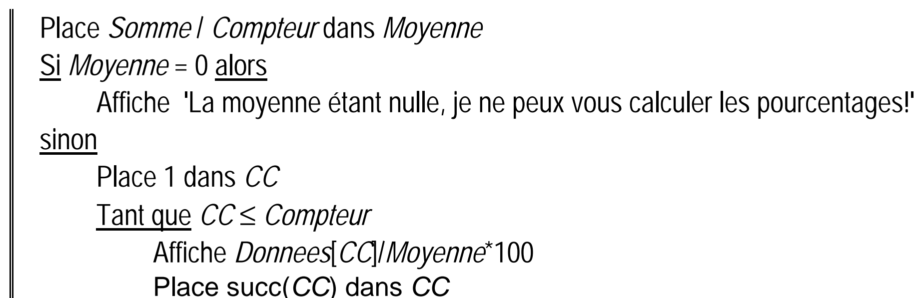
Il suffit de reprendre la portion grisée et encadrée d'accolades du GNS proposé



pour l'intégrer dans une alternative :



ou encore



C'est un bon réflexe, à chaque fois qu'on commande une division, de s'interroger sur la certitude qu'on a qu'à cet endroit du programme le diviseur (= le nombre par lequel on divise) est bien différent de 0.

2.5 Comment dire

Et d'abord la version (presque) nue du programme :

```
program MOYENNE_ET_COMPARAISONS;
uses WinCRT;
var
    Donnees : array[1..100] of real;
    X,
    Somme,
    Bidon,
    Moyenne : real;
```

```
    Compteur,
    CC : integer;

begin
readln(Bidon);
Compteur:=0;
Somme:=0;
readln(X);
while X <> Bidon do
   |begin
   |Compteur:=Compteur+1;
   |Somme:=Somme+X;
   |Donnees[Compteur]:=X;
   |readln(X);
   |end;
if Compteur=0 then
   |writeln('Vous ne m''avez pas fourni de données !')
else
   |begin
   |Moyenne:=Somme/Compteur;
   |writeln('La moyenne est de ',Moyenne:12:3);
   |if Moyenne=0 then
   |   |writeln('La moyenne étant nulle, je ne peux vous calculer les pourcentages!')
   |else
   |   |begin
   |   |CC:=1;
   |   |while CC <= Compteur do
   |   |     |begin
   |   |     |writeln('Donnée ',CC,' : ',Donnees[CC]:6:1,'; Pourcentage : ',
   |   |     |          Donnees[CC]/Moyenne*100:3:0);
   |   |     |CC:=succ(CC);
   |   |     |end;
   |   |end;
   |end;
end.
```

Et la version habillée :

```
program MOYENNE_ET_COMPARAISONS;
uses WinCRT;
(* Il fait calculer la moyenne d'une série d'au plus 100 données (terminée par une donnée
   bidon préalablement choisie par l'utilisateur) puis fait indiquer, pour
   chacune des données le pourcentage qu'elle représente par rapport à cette moyenne. *)
(* Il n'y a aucune vérification du fait que l'utilisateur fournit au plus 100 données *)

var
    Donnees : array[1..100] of real;                                    (1)
    (* étagère à 100 tiroirs de type réel qui contiendra les diverses données lues *)
    X,
    (* qui contiendra chaque donnée avant son placement dans l'étagère *)
    Somme,
    (* qui contiendra la somme des données *)
    Bidon,
    (* qui contiendra la donnée bidon *)
    Moyenne
    (* qui contiendra la moyenne des données *)
    : real;
    Compteur,
    (* pour compter les données lues *)
    CC
    (* pour explorer l'étagère lors de l'affichage *)
    : integer;

begin
clrscr; (* pour faire effacer l'écran *)
(* avertissement pour l'utilisateur *)
writeln('Vous allez me fournir une série de données réelles et je vous');
writeln('en fournirai la moyenne, puis, pour chaque donnée, le pourcentage');
writeln('représenté par rapport à cette moyenne.');
writeln('Attention! Vous avez droit à 100 données au plus.');
writeln('Donnez-moi d''abord la donnée avec laquelle vous terminerez');
write('tantôt la liste : ');
readln(Bidon);
```

```
writeln;writeln; (* pour passer deux lignes *)
(* initialisations *)
Compteur:=0;
Somme:=0;
(* lecture de la première donnée *)
write('Donnée : ');
readln(X);
(* traitements et lectures *)
while X <> Bidon do
    begin
    Compteur:=Compteur+1;
    Somme:=Somme+X;
    Donnees[Compteur]:=X;                                    (2)
    write('Donnée suivante : ');
    readln(X);
    end;
if Compteur=0 then
    writeln('Vous ne m''avez pas fourni de données !')
else
    begin
    Moyenne:=Somme/Compteur;
    writeln('La moyenne est de ',Moyenne:12:3);
    if Moyenne=0 then
        writeln('La moyenne étant nulle, je ne peux vous calculer les pourcentages!')
    else
        begin
        CC:=1;
        while CC <= Compteur do
            begin
            writeln('Donnée ',CC,' : ',Donnees[CC]:6:1,'; Pourcentage : ',
                     Donnees[CC]/Moyenne*100:3:0);               (3)
            CC:=succ(CC);
            end;
        end;
    end;
readln;
end.
```

Quelques commentaires :

(1) On retrouve la manière de déclarer un tableau (étagère) en Pascal :

```
identificateur de variable : array[1ère étiquette..dernière étiquette] of
type des tiroirs
```

Nous y reviendrons plus en détail dans la suite.

(2), (3) Pour accéder à un tiroir particulier d'une étagère, soit pour y placer une donnée, soit pour en consulter le contenu, la syntaxe est la suivante :

```
identificateur de l'étagère[valeur de l'étiquette]
```

cette valeur pouvant être écrite comme une constante, (le contenu d')une variable ou une expression.

3. Quelques compléments théoriques sur les types primitifs de données en Pascal

Avant d'aborder le concept central du présent chapitre, celui de tableau (étagère), voici quelques compléments à propos des types de données manipulées par l'exécutant-ordinateur. Nous connaissons déjà les entiers (integer), les réels (real) et les chaînes de caractères (string). Nous allons examiner quelques types supplémentaires.

En cours, j'appelle ce type d'épisode un cours "démonstration tupperware" : la seule visée, c'est de dresser une liste de possibilités, sans plus. Nous n'allons pas vraiment utiliser immédiatement, au sein de problèmes à résoudre, les possibilités mentionnées, mais seulement les lister. C'est on ne peut plus fastidieux...

3.1 Codage d'entiers

3.1.1 Les divers types entiers

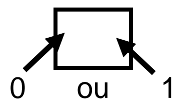
A côté du type integer déjà connu, il nous faut ajouter d'autres types de données (et les types de variables qui peuvent les accueillir). Il existe cinq types d'entiers prédéfinis. A chacun correspond un sous-intervalle de l'ensemble des entiers :

Type	Intervalle des entiers représentables	Taille
shortint	-128..127	8 bits (1 octet)
integer	-32768..32767	16 bits (2 octet)
longint	-2147483648..2147483647	32 bits (4 octet)
byte	0..255	8 bits (1 octet)
word	0..65535	16 bits (2 octet)

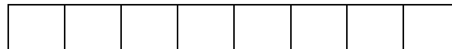
On notera que dans les types byte et word, seuls des entiers positifs sont représentables.

Les entiers manipulés et les variables pouvant les accueillir correspondent évidemment à des "cases" de la mémoire centrale d'un ordinateur. Vous savez peut-être que la plus petite entité constituant cette mémoire est le bit, case ne pouvant prendre que deux valeurs, qu'on désigne habituellement par les symboles 0 et 1. L'alphabet de l'ordinateur est donc binaire, puisque tout ce qui prend place en mémoire est finalement codé (représenté) sous le forme d'une suite de symboles 0 et 1.

La mémoire centrale peut donc être modélisée comme une immense juxtaposition de cases élémentaires, chacune ne pouvant comporter que l'un des symboles 0 ou 1 :



Si le bit est le constituant ultime de la mémoire, cette dernière est en général organisée comme une suite de groupes de 8 bits, qu'on appelle octet (ou byte, en anglais).



Tout ce que l'ordinateur manipule est donc codé, représenté, sur un ou plusieurs octets.

Ainsi, une variable entière de type INTEGER correspond exactement à deux octets, donc à 16 bits. Comme on veut pouvoir sur ces deux octets représenter des nombres aussi bien négatifs que positifs, on va consacrer 1 bit à coder le signe du nombre[2] et il restera donc 15 octets pour coder le nombre lui-même, en binaire évidemment.

bit utilisé pour coder le signe du nombre

Avec un bit on peut coder en binaire deux nombres seulement : 0 et 1;
avec 2 bits, on multiplie par deux la quantité de nombres qu'on peut écrire : il y a les 4 configurations 00 (0), 01 (1), 10 (2) et 11 (3).
Il est facile de voir qu'avec N bits, on pourra écrire en binaire 2^N nombres, soient les nombres entre 0 et 2^N-1.

Ainsi donc, avec 15 bits, on pourra écrire les 2^{15} nombres compris entre 0 et 2^{15}-1, soit entre 0 et 32767. Il y aura ensuite tous les nombres qu'on pourra aussi écrire avec le premier bit signalant qu'ils sont négatifs. Comme 0 est déjà compté avec les positifs, on pourra avec les 15 bits coder 2^{15} nombres négatifs allant de -1 à -2^{15} soit de -1 à -32768.

Ainsi, un entier de type integer, signé et codé sur deux octets, pourra être compris entre - 32768 et 32767.

2 C'est en réalité un peu plus compliqué que cela, mais, au niveau des conséquences, c'est à ce bit perdu pour le signe que ça revient.

Il est facile de voir que les autres codages d'entiers, avec ou sans signe, sur 1, 2 ou 4 octets conduisent aux valeurs données plus haut.

3.1.2 Attention aux dépassements dangereux

Voici un court programme en Pascal :

```
program DEMO;
uses WinCRT;
var A,B : integer;
begin
A:=20000;
B:=20000;
writeln('La somme de ',A,' et ',B,' est ',A+B);
end.
```

Et voici ce que donne son exécution :

```
La somme de 20000 et 20000 est -25536
```

Le résultat correct, 40.000, qui n'est pas codable en tant que integer est remplacé par un nombre négatif, -25.536 ! (Pourquoi à votre avis ?)

Bien évidemment, l'affectation à une variable d'une quantité constante qui n'est plus dans l'intervalle des valeurs admissibles pour ce type de variable, provoquera une erreur lors de la traduction du programme Pascal. Ainsi :

```
program DEMO;
uses WinCRT;
var A: integer;
begin
A:=40000;
end.
```

conduit à la détection d'une erreur lors de la compilation

? Pourquoi n'y a-t-il pas eu erreur à la compilation dans le premier programme présenté ci-dessus ?

Il faut retenir de ceci que, quel que soit le type d'entiers utilisé, seul un intervalle des entiers est manipulable par l'exécutant ordinateur. Il n'y a qu'en mathématique que les entiers constituent un ensemble infini...

3.2 Les réels

3.2.1 Codage des réels

De la même manière, les nombres réels et les variables qui permettent de les stocker peuvent en Pascal être de différents types :

Type	Intervalle (approximatif)	Chiffres significatifs	Taille en octets
single	$-3.4*10^{38}$ .. $-1.5*10^{-45}$ et $1.5*10^{-45}$ .. $3.4*10^{38}$	7-8	4
real	$-1.7*10^{38}$ .. $-2.9*10^{-39}$ et $2.9*10^{-39}$ .. $1.7*10^{38}$	11-12	6
double	$-1.7*10^{308}$ .. $-5.0*10^{-324}$ et $5.0*10^{-324}$ .. $1.7*10^{308}$	15-16	8
extended	$-1.1*10^{4932}$ .. $-3.4*10^{-4932}$ et $3.4*10^{-4932}$ .. $1.1*10^{4932}$	19-20	10
comp	$-9.2*10^{18}$ .. $9.2*10^{18}$	19-20	8 (entier)

Je n'entrerai pas ici dans des détails sur la manière dont les réels sont codés : ils sont en général représentés en deux parties : la *mantisse*, reprenant les chiffres significatifs et l'*exposant* reprenant la puissance de 10. Ce codage mantisse - exposant apparaît clairement lorsqu'on demande

l'affichage d'un réel. Voici un court programme montrant ce type d'affichage et mettant en évidence le nombre de chiffres significatifs permis par les différents types réels. Il demande simplement l'affichage de la valeur 1/3, emmagasinée dans des variables de types single, real, double et extended :

```
program DEMO;
uses WinCrt;
var   A : single;
      B : real;
      C : double;
      D : extended;

begin
clrscr;
A:=1/3;
B:=1/3;
C:=1/3;
D:=1/3;
writeln('A (single) : ':20,A);    (1)
writeln('B (real) : ':20,B);
writeln('C (double) : ':20,C);
writeln('D (extended): ':20,D);
end.
```

qui conduit aux résultats suivants :

```
        A (single) :   3.33333343267441E-0001
          B (real) :   3.33333333333485E-0001
        C (double) :   3.33333333333333E-0001
      D (extended):   3.33333333333333E-0001
```

- On notera que le nombre de chiffres corrects dépend du type de codage envisagé; on notera aussi l'écriture utilisée :

  3.33333343267441E-0001 signifie $3.33333343267441 \times 10^{-1}$ soit 0.333333343267441

- Un détail à propos du programme : on note dans les instructions d'affichage (comme en **(1)**) que des données (constantes) de type chaînes de caractères sont suivies de **:20**. Ceci est à rapprocher de ce qui a été dit pour les réels en page 17. Lorsqu'on fait suivre une valeur de type chaîne de deux points puis d'une constante entière lors de l'affichage, on précise ainsi combien de caractères au total seront pris pour l'affichage de la chaîne, cette dernière étant cadrée à droite (= des blancs sont, s'il le faut, rajoutés devant).

3.2.2 Attention à la précision lors du codage des réels

Les nombres réels sont donc codés en mémoire avec une précision limitée, dépendant du type de codage choisi. Voici un court programme qui illustre parfaitement ce problème de précision.

```
program DEMO;
uses WinCrt;
var A,B : single;
    C,D : double;
begin
clrscr;
A:=123456789999.0;
B:=123456789000.0;
C:=123456789999.0;
D:=123456789000.0;
writeln('La différence de 123456789999.0 et 123456789000.O est ',A-B);
writeln('La différence de 123456789999.0 et 123456789000.0 est ',C-D);

end.
```

On y fait placer dans deux variables single (*A* et *B*) deux "gros" nombres réels différents de 999; on fait ensuite de même avec deux variables de type double (*C* et *D*). Dans chaque cas, on fait ensuite afficher la différence (qui devrait être de 999).

Et voici les résultats :

```
La différence de 123456789999.0 et 123456789000.0 est  0.00000000000000E+0000
La différence de 123456789999.0 et 123456789000.0 est  9.99000000000000E+0002
```

Dans le cas du codage double, la différence annoncée est bien de $9.99x10^2$ soit 999; mais avec le codage single, la différence annoncée est de 0 et les deux valeurs sont considérées comme identiques.

Il est bon de retenir que le codage des réels se fait donc avec une précision dépendant du type retenu, mais toujours limitée. Les erreurs causées par cette précision limitée sont appelées **erreurs de troncature**.

3.3 Les outils de manipulation des nombres

3.3.1 Les opérations

Il y a d'abord les outils qui, sur base de deux nombres (les opérands), en fournissent un troisième (le résultat). Ces outils sont des **opérations**, comme la somme ou le produit, et la notation en est habituellement infixée; c'est à dire qu'on écrit A + B et pas + (A, B).

Les voici sous forme d'un tableau donnant les types possibles d'opérands et le type de résultat :

Opérateur	Opération	Types des opérands	Type du résultat
+	Addition	entier réel	entier réel
-	Soustraction	entier réel	entier réel
*	Multiplication	entier réel	entier réel
/	Division	entier réel	réel réel
div	Division entière	entier	entier
mod	Reste	entier	entier

3.3.2 Les fonctions numériques

Ces outils diffèrent seulement en ce que, sur base d'un nombre, ils en fournissent un autre, et par le fait que la notation utilisée est préfixée. Un tel outil est une **fonction** : sur base d'un argument, la fonction fournit un résultat.

Voici les fonctions numériques disponibles en Pascal :

Fonction	Syntaxe	Argument	Résultat
Valeur absolue	Abs	entier ou réel	comme l'argument abs(-3.56) est 3.56, abs(5) est 5
Arc tangente	ArcTan	réel	réel
Cosinus	Cos	réel	réel
Exponentielle	Exp	réel	réel
Logarithme naturel	Ln	réel	réel

 © CeFIS - FUNDP 2002

Impair	Odd	entier (LongInt)	booléen (voir plus loin) odd(40000) est false (faux)
Au hasard	Random	entier (Word)	entier au hasard ≥ 0 et $<$ argument random(6) désigne un entier au hasard entre 0 et 5
Sinus	Sin	réel	réel
Carré	Sqr	entier ou réel	comme l'argument
Racine carrée	Sqrt	réel	réel

Il y a également plusieurs fonctions liées au passage des réels aux entiers (arrondis et troncatures)

Fonction	**Syntaxe**	**Argument**	**Résultat**
Arrondi à l'entier le plus proche	Round	réel	entier (LongInt) round(-1.5) est -2, round(-1.4) est -1, round(1.5) est 2, round(1.4) est 1
Troncature pour ne garder que la partie entière	Trunc	réel	entier (LongInt) trunc(-1.5) est -1, trunc(-1.4) est -1, trunc(1.5) est 1, trunc(1.4) est 1
Partie entière pour éliminer du réel la partie décimale	Int	réel	**réel** int(-1.5) est -1.0, int(-1.4) est -1.0, int(1.5) est 1.0, int(1.4) est 1.0
Partie fractionnaire ne retient que la partie fractionnaire	Frac	réel	réel frac(-1.5) est -0.5, int(-1.4) est -0.4, int(1.5) est 0.5, int(1.4) est 0.4

3.3.3 Opérations, fonctions, c'est finalement la même chose

Vous pouvez passer cette partie, si vous n'aimez pas trop la théorie.

Effectuer la somme de deux nombres, prendre le carré d'un nombre ou calculer son cosinus, c'est finalement établir des correspondances :

+ : RxR ⟶ R	carré : R ⟶ R	cos : R ⟶ R
(x,y) ⟶ x+y	x ⟶ x^2	x ⟶ cos(x)

R désigne ici l'ensemble des nombres réels et RxR l'ensemble des couples de nombres réels.

Ce qu'il y a de commun entre tous ces exemples, c'est qu'à chaque fois, à un élément d'un ensemble (l'argument de la fonction) on fait correspondre un élément d'un autre ensemble (ou du même) (le résultat).

Il arrive que l'ensemble de départ soit un ensemble de couples (ce qu'on appelle un produit cartésien) : dans ce cas la fonction est quelquefois appelée opération et le résultat est (mais pas toujours) noté de la façon habituelle pour les opérations : x+y, x*y, x-y, x/y,...Mais on a aussi x^y.

Dans beaucoup de cas l'ensemble de départ (où se trouve l'argument de la fonction) est un ensemble simple, comme lorsqu'on parle de x^2 , de cos(x) de int(x), etc.

Enfin, on insiste souvent plus sur le résultat apporté par la fonction (ou l'opération) que sur la correspondance établie : au lieu des flèches qui marquent la correspondance on se contente de noter x^2 ou cos(x) ou x*y.

3.4 *Les caractères*

Nous savons déjà que l'exécutant-ordinateur est capable de manipuler des données de type chaînes de caractères (string). Il peut évidemment manipuler des caractères isolés et pourra disposer de variables susceptibles d'accueillir l'un ou l'autre de ces caractères.

Pour déclarer de telles variables, on écrira

```
var ... : char;
```

3.4.1 Les constantes de type caractère

La liste des caractères manipulés dépend de la version de Pascal utilisée, ou plus exactement du système d'exploitation dans lequel l'environnement de programmation Pascal s'insère.

On devine bien entendu que, comme les nombres, les caractères seront codés en mémoire sous forme d'un ou plusieurs octets. Pendant longtemps et actuellement encore, on utilisait un octet pour coder un caractère. Ceci donnait évidemment droit au codage de 256 caractères différents puisque on peut avoir 2^8 soit 256 configurations différentes pour un octet.

A chaque caractère est alors associé un nombre entre 0 et 255 et ce nombre est codé en binaire sur un octet. Malheureusement, le choix des caractères retenus, comme celui des nombres qui leur sont associés, n'est pas unique ou uniforme. Pendant longtemps, c'est le code dit ASCII (American Standard Code for Information Interchange) qui a été en vigueur. Voici la succession des caractères retenus et leurs numéros, dans ce code ASCII primitif :

<table>
<tr><td>espace</td><td>32</td><td>0</td><td>48</td><td>A</td><td>65</td><td>[</td><td>91</td><td>a</td><td>97</td><td>{</td><td>123</td></tr>
<tr><td>!</td><td>33</td><td>1</td><td>49</td><td>B</td><td>66</td><td>\</td><td>92</td><td>b</td><td>98</td><td>|</td><td>124</td></tr>
<tr><td>"</td><td>34</td><td>2</td><td>50</td><td>C</td><td>67</td><td>]</td><td>93</td><td>c</td><td>99</td><td>}</td><td>125</td></tr>
<tr><td>#</td><td>35</td><td>3</td><td>51</td><td>D</td><td>68</td><td>^</td><td>94</td><td>d</td><td>100</td><td>~</td><td>126</td></tr>
<tr><td>$</td><td>36</td><td>4</td><td>52</td><td>E</td><td>69</td><td>_</td><td>95</td><td>e</td><td>101</td><td></td><td></td></tr>
<tr><td>%</td><td>37</td><td>5</td><td>53</td><td>F</td><td>70</td><td>`</td><td>96</td><td>f</td><td>102</td><td></td><td></td></tr>
<tr><td>&</td><td>38</td><td>6</td><td>54</td><td>G</td><td>71</td><td></td><td></td><td>g</td><td>103</td><td></td><td></td></tr>
<tr><td>'</td><td>39</td><td>7</td><td>55</td><td>H</td><td>72</td><td></td><td></td><td>h</td><td>104</td><td></td><td></td></tr>
<tr><td>(</td><td>40</td><td>8</td><td>56</td><td>I</td><td>73</td><td></td><td></td><td>i</td><td>105</td><td></td><td></td></tr>
<tr><td>)</td><td>41</td><td>9</td><td>57</td><td>J</td><td>74</td><td></td><td></td><td>j</td><td>106</td><td></td><td></td></tr>
<tr><td>*</td><td>42</td><td></td><td></td><td>K</td><td>75</td><td></td><td></td><td>k</td><td>107</td><td></td><td></td></tr>
<tr><td>+</td><td>43</td><td></td><td></td><td>L</td><td>76</td><td></td><td></td><td>l</td><td>108</td><td></td><td></td></tr>
<tr><td>,</td><td>44</td><td>:</td><td>58</td><td>M</td><td>77</td><td></td><td></td><td>m</td><td>109</td><td></td><td></td></tr>
<tr><td>-</td><td>45</td><td>;</td><td>59</td><td>N</td><td>78</td><td></td><td></td><td>n</td><td>110</td><td></td><td></td></tr>
<tr><td>.</td><td>46</td><td><</td><td>60</td><td>O</td><td>79</td><td></td><td></td><td>o</td><td>111</td><td></td><td></td></tr>
<tr><td>/</td><td>47</td><td>=</td><td>61</td><td>P</td><td>80</td><td></td><td></td><td>p</td><td>112</td><td></td><td></td></tr>
<tr><td></td><td></td><td>></td><td>62</td><td>Q</td><td>81</td><td></td><td></td><td>q</td><td>113</td><td></td><td></td></tr>
<tr><td></td><td></td><td>?</td><td>63</td><td>R</td><td>82</td><td></td><td></td><td>r</td><td>114</td><td></td><td></td></tr>
<tr><td></td><td></td><td>@</td><td>64</td><td>S</td><td>83</td><td></td><td></td><td>s</td><td>115</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>T</td><td>84</td><td></td><td></td><td>t</td><td>116</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>U</td><td>85</td><td></td><td></td><td>u</td><td>117</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>V</td><td>86</td><td></td><td></td><td>v</td><td>118</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>W</td><td>87</td><td></td><td></td><td>w</td><td>119</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>X</td><td>88</td><td></td><td></td><td>x</td><td>120</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>Y</td><td>89</td><td></td><td></td><td>y</td><td>121</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>Z</td><td>90</td><td></td><td></td><td>z</td><td>122</td><td></td><td></td></tr>
</table>

et étendu

Ç	128	û	150	¼	172	┬	194	╪	216	ε	238
ü	129	ù	151	¡	173	├	195	┘	217	∩	239
é	130	ÿ	152	«	174	─	196	┌	218	≡	240
â	131	Ö	153	»	175	┼	197	█	219	±	241
ä	132	Ü	154	░	176	╞	198	▄	220	≥	242
à	133	¢	155	▒	177	╟	199	▌	221	≤	243
å	134	£	156	▓	178	╚	200	▐	222	⌠	244
ç	135	¥	157	│	179	╔	201	▀	223	⌡	245
ê	136	₧	158	┤	180	╩	202	α	224	÷	246
ë	137	ƒ	159	╡	181	╦	203	ß	225	≈	247
è	138	á	160	╢	182	╠	204	Γ	226	°	248
ï	139	í	161	╖	183	═	205	π	227	∙	249
î	140	ó	162	╕	184	╬	206	Σ	228	·	250
ì	141	ú	163	╣	185	╧	207	σ	229	√	251
Ä	142	ñ	164	║	186	╨	208	µ	230	^{n}	252
Å	143	Ñ	165	╗	187	╤	209	τ	231	^{2}	253
É	144	ª	166	╝	188	╥	210	Φ	232	■	254
æ	145	º	167	╜	189	╙	211	Θ	233		255
Æ	146	¿	168	╛	190	╘	212	Ω	234		
ô	147	⌐	169	┐	191	╒	213	δ	235		
ö	148	¬	170	└	192	╓	214	∞	236		
ò	149	½	171	┴	193	╫	215	φ	237		

On notera que les caractères portant les numéros 0 à 31 ne sont pas repris dans ce tableau. On pourra consulter à ce propos : "DUCHATEAU C. : Initiation à l'informatique" (téléchargeable à l'adresse http://www.det.fundp.ac.be/cefis/publications/charles/ini-5-51.pdf.

Le code ASCII étendu, qui était la règle jusque dans les années 90 (avec les ordinateurs tournant sous le système d'exploitation MS-DOS) a été ensuite remplacé par un autre système de codage : le code ANSI (CP 850 pour les spécialistes) (avec les PC tournant sous Windows).

Dans ce code, les 128 premiers caractères restent les mêmes que pour ASCII, mais les 128 derniers sont différents.

	128	–	150	¬	172	Â	194	Ø	216	î	238
	129	—	151	-	173	Ã	195	Ù	217	ï	239
‚	130	˜	152	®	174	Ä	196	Ú	218	ð	240
ƒ	131	™	153	¯	175	Å	197	Û	219	ñ	241
„	132	š	154	°	176	Æ	198	Ü	220	ò	242
…	133	›	155	±	177	Ç	199	Ý	221	ó	243
†	134	œ	156	^{2}	178	È	200	Þ	222	ô	244
‡	135		157	^{3}	179	É	201	ß	223	õ	245
ˆ	136		158	´	180	Ê	202	à	224	ö	246
‰	137	Ÿ	159	µ	181	Ë	203	á	225	÷	247
Š	138		160	¶	182	Ì	204	â	226	ø	248
‹	139	¡	161	·	183	Í	205	ã	227	ù	249
Œ	140	¢	162	¸	184	Î	206	ä	228	ú	250
	141	£	163	^{1}	185	Ï	207	å	229	û	251
	142	¤	164	º	186	Ð	208	æ	230	ü	252
	143	¥	165	»	187	Ñ	209	ç	231	ý	253
	144	¦	166	¼	188	Ò	210	è	232	þ	254

‘	145	§	167	½	189	Ó	211	é	233	ÿ	255
’	146	¨	168	¾	190	Ô	212	ê	234		
“	147	©	169	¿	191	Õ	213	ë	235		
”	148	ª	170	À	192	Ö	214	ì	236		
•	149	«	171	Á	193	×	215	í	237		

Enfin, un dernier système de codage, l'UNICODE est en train de s'imposer. Comme les caractères y sont codés sur deux octets, ce système permet le codage de 2^{16} soit 65536 caractères différents.

Rappelons qu'au sein d'un programme Pascal les constantes de type caractère doivent figurer entre apostrophes : 'a', '#', ...

3.4.2 Les outils de traitement des caractères

On a d'abord deux fonctions :

$$\text{chr : } [0,255] \longrightarrow \mathcal{C} \qquad \text{et} \qquad \text{ord : } \mathcal{C} \longrightarrow [0,255]$$
$$n \longrightarrow \text{chr(n)} \qquad\qquad c \longrightarrow \text{ord(c)}$$

[0,255] désignant l'ensemble des entiers entre 0 et 255 et $\mathcal{C}$ l'ensemble des caractères retenus.

chr(n) est le caractère dont le numéro est n (dans ASCII ou ANSI) et ord(c) est le numéro du caractère c; les deux fonctions ord et chr sont donc réciproques l'une de l'autre.

Deux autres fonctions, prédécesseur et successeur, sont aussi disponibles :

$$\text{pred: } \mathcal{C} \longrightarrow \mathcal{C} \qquad \text{et} \qquad \text{succ: } \mathcal{C} \longrightarrow \mathcal{C}$$
$$c \longrightarrow \text{pred(c)} \qquad\qquad c \longrightarrow \text{succ(c)}$$

Le prédécesseur fournit évidemment le caractère précédent et le successeur le caractère suivant, l'ordre étant celui donné par la numérotation du codage envisagé.

A noter encore que les données de type caractère peuvent être, comme les nombres, comparées par =, <, >, ≤, ≥, ≠ (respectivement notés en Pascal : =, <, >, <=, >=, <>), l'ordre étant toujours donné par le codage retenu. Chacune de ces comparaisons conduit à une condition, ce que nous appellerons dans un moment une "expression booléenne".

3.5 Le type booléen

Nous avons déjà, sans le savoir, utilisé le type booléen. Rappelons-nous que les conditions que peut tester l'exécutant-ordinateur sont toujours des comparaisons ou des combinaisons de comparaisons liées par les mots and (et), or (ou) ou not (non). Chaque fois que nous avons écrit une comparaison, nous avons fait manipuler par l'exécutant une expression de nature booléenne.

> Pour la plupart des types de données, on commence par présenter les constantes, puis les variables et on termine par les expressions. En ce qui concerne les booléens, c'est dans un autre ordre que les concepts ont été abordés : on a d'abord parlé de condition, c'est à dire d'expression de type booléen; il reste à parler des constantes et des variables de ce type.

3.5.1 Les constantes booléennes

Elles sont seulement au nombre de deux : false (faux) et true (vrai). Ces constantes sont ordonnées avec false < true.

Il faut se garder de confondre les deux constantes false et true avec les deux chaînes de caractères 'false' et 'true'. Les constantes booléennes false et true ne sont ni lisibles au clavier, ni affichables à l'écran.

3.5.2 Les variables booléennes

Comme le type booléen existe, on peut évidemment définir des variables de ce type. Ceci se fait par :

```
var ... : boolean;
```

On peut bien entendu affecter à une variable booléenne une valeur de type booléen. Ainsi, si on a

```
var    b1, b2: boolean;
       Compteur : integer;
```

on pourra écrire

```
b1 := Compteur < 10
b1:= b1 or not b2
if b1 then...
b2 := false
```

Redisons qu'on ne peut donc ni faire imprimer le contenu d'une variable booléenne, ni non plus la remplir par lecture.

3.5.3 Les outils donnant un résultat booléen et les expressions booléennes

Nous connaissons déjà les expressions booléennes : ce sont ce que nous avions appelé jusqu'ici les conditions. Simplement, nous allons pouvoir dans ces expressions utiliser également des variables booléennes. Si on déclare

```
var    b1, b2: boolean;
       i1, i2 : integer;
       c: char;
```

alors

```
b1 and (i1 <> i2)
b1 or (ord(c)>i2)
b2 and not(b1 or (i1=i2))
odd(i1-i2) or (b1=b2)
```

sont des expressions booléennes (= des conditions).

Les outils fournissant un résultat booléen sont les comparaisons : =, <, >, ≤, ≥, ≠. Ce sont bien entendu en réalité des fonctions. Par exemple :

$$< : \mathbb{R}\times\mathbb{R} \longrightarrow \mathcal{B} \qquad (x,y) \longrightarrow x<y$$

$$= : \mathbb{Z}\times\mathbb{Z} \longrightarrow \mathcal{B} \qquad (m,n) \longrightarrow m=n$$

$$< : \mathcal{C}\times\mathcal{C} \longrightarrow \mathcal{B} \qquad (c,d) \longrightarrow c<d$$

$\mathcal{B}$ désignant l'ensemble {false, true} des constantes booléennes, $\mathbb{Z}$ celui des entiers (relatifs) (en tous cas ceux représentables en mémoire) et $\mathbb{R}$ l'ensemble des réels (codables en mémoire).

Rappelons (pour souffler un peu) que tout ceci est à la programmation, ce que la consultation d'un catalogue d'ustensiles de cuisine est à l'activité de cuisiner.

On a également les outils bien connus et rappelés ci-dessus qui sur base de valeurs booléennes fournissent une valeur booléenne : and, or et not.

and : $B \times B \longrightarrow B$; $(b,d) \longrightarrow$ b and d | or : $B \times B \longrightarrow B$; $(b,d) \longrightarrow$ b or d | not : $B \longrightarrow B$; b $\longrightarrow$ not b

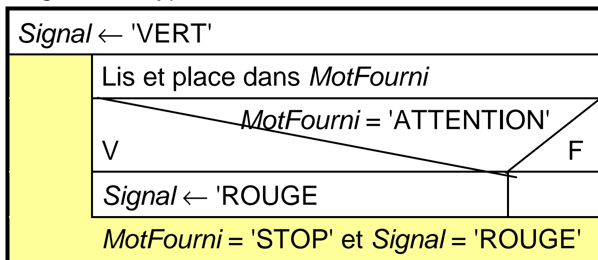
3.5.4 Un petit problème

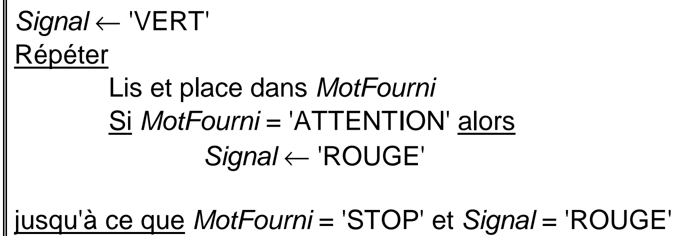
3.5.4.1 Rappel

Nous avons eu l'occasion, lors du traitement du problème de l'arrêt à STOP à condition que ATTENTION ait été reçu auparavant (1ère partie, pages 199 et suivantes) d'écrire :

MotFourni : contiendra successivement les mots fournis, de type chaîne

Signal : de type chaîne





conduisant au programme suivant :

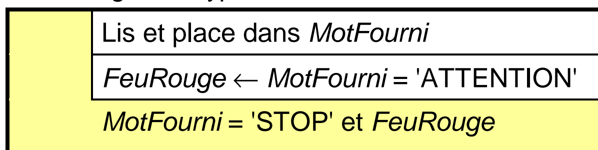
```
program ARRET;
uses WinCrt;
var    MotFourni : string[30]; (* variable chaîne d'au plus 30 caractères *)
       Signal : string[5]; (* variable chaîne d'au plus 5 caractères *)
begin
Signal := 'VERT';
repeat
       readln(MotFourni);
       if MotFourni = 'ATTENTION' then
              Signal := 'ROUGE';
until (MotFourni = 'STOP') and (Signal = 'ROUGE');
end.
```

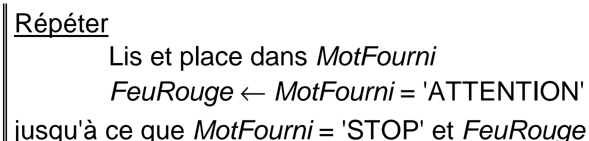
3.5.4.2 Une solution alternative

On peut, en utilisant le type booléen proposer la solution suivante :

MotFourni : contiendra successivement les mots fournis, de type chaîne

FeuRouge : de type **booléen**





conduisant à

```
program ARRET;
uses WinCrt;
var    MotFourni : string[30];
       FeuRouge : boolean;
begin
repeat
       readln(MotFourni);
       FeuRouge := MotFourni = 'ATTENTION';
until (MotFourni = 'STOP') and FeuRouge;
end.
```

? Cette seconde solution vous semble-t-elle syntaxiquement correcte ? Conduit-elle à une exécution correcte ?

3.5.4.3 *Commentaires*

- La solution proposée est correcte sur le plan de la syntaxe. En effet,

  FeuRouge étant booléen, on peut écrire

  FeuRouge ← *MotFourni* = 'ATTENTION'

  puisque *MotFourni* = 'ATTENTION' est de type booléen. On peut aussi écrire

  jusqu'à ce que *MotFourni* = 'STOP' et *FeuRouge*

  pour la même raison.

- Par contre, le programme conduit à un traitement erroné. En effet, **à chaque passage** par l'instruction

  FeuRouge ← *MotFourni* = 'ATTENTION'

  on place dans *FeuRouge* le résultat de la comparaison *MotFourni* = 'ATTENTION' et donc, si après avoir reçu ATTENTION on reçoit autre chose, comme STOP par exemple, on va placer faux dans *FeuRouge*; et lors du test

  jusqu'à ce que *MotFourni* = 'STOP' et *FeuRouge*

  on poursuivra, bien que ATTENTION ait été lu et STOP également.

3.5.4.4 *Une autre proposition*

Que pensez vous de :

Lis et place dans *MotFourni*
FeuRouge ← *FeuRouge* ou *MotFourni* = 'ATTENTION'
MotFourni = 'STOP' et *FeuRouge*

```
Répéter
   Lis et place dans MotFourni
   FeuRouge ← FeuRouge ou MotFourni =
                    'ATTENTION'
jusqu'à ce que MotFourni = 'STOP' et FeuRouge
```

3.6 *Les types scalaires*

C'est ainsi que l'on désigne en Pascal l'ensemble des types booléens, caractères et les divers types entiers. Comme on aura l'occasion de le constater, ils sont particulièrement importants sur le plan conceptuel.

Les trois fonctions suivantes existent pour tous les types scalaires :

succ : $\mathcal{S} \longrightarrow \mathcal{S}$	pred : $\mathcal{S} \longrightarrow \mathcal{S}$	ord : $\mathcal{S} \longrightarrow \mathcal{L}$
$x \longrightarrow$ succ(x)	$x \longrightarrow$ pred(x)	$x \longrightarrow$ ord(x)

ou $\mathcal{S}$ représente l'un quelconque des types scalaires : boolean, char, integer, word, byte, shortint et longint et $\mathcal{L}$ représente l'ensemble des entiers de type longint.

Ainsi, par exemple :

succ(45) vaut 46	succ('b') est 'c'	succ(false) vaut true	succ(true) est indéterminé
pred((45) vaut 44	pred('b') est 'a'	pred(true) vaut false	pred(false) est indéterminé
ord(false) est 0	ord('A') est 65	ord(66666) est 66666	ord(-10) est -10

Comme nous le verrons, le fait de pouvoir écrire "**au suivant !**" pour les valeurs de type scalaire est essentiel, comme l'est, en conséquence, le fait que lorsqu'on donne deux valeurs d'un

même type scalaire, l'exécutant est capable de déterminer les valeurs qui se succèdent entre les deux valeurs précisées.

3.7 *Les types intervalles*

Pascal offre la possibilité de définir à partir d'un type scalaire un "sous-type" qui ne reprend qu'une partie des constantes du type concerné. On appelle type **intervalle** un type défini de cette manière. On va ainsi, en quelque sorte, mettre en évidence un intervalle particulier parmi les entiers ou les caractères, en nommant cet intervalle particulier.

3.7.1 Des exemples

On pourra par exemple écrire en début d'un programme Pascal :

`type Face = 1..6;` (intervalle parmi les entiers) ou encore

`type Majuscules = 'A'..'Z';` (intervalle parmi les caractères) ou encore

`type Chiffres = 0..9;` (intervalle parmi les entiers)

3.7.2 Quelques commentaires

C'est le mot réservé type qui permet la définition d'un type nouveau (ici un type intervalle) :

- Il faut bien percevoir que les termes Faces, Majuscules ou Chiffres utilisés pour désigner des types dans les exemples ci-dessus sont de même nature que les mots INTEGER ou char; définir un type intervalle, c'est en quelque sorte préciser les constantes qui feront partie de ce type.
- Il ne faut pas confondre la définition d'un type et celle de variables. On pourra par exemple avoir

  `type Face = 1..6;` puis

  `var De : Face;`

  De est une variable de type Face; on ne pourra donc y faire placer que des entiers entre 1 et 6. Toute tentative d'y faire déposer autre chose conduira à une erreur.
- Nous verrons dans l'avenir d'autres possibilités de définir des types, qui ne seront plus seulement des intervalles d'un type scalaire (comme les types énumérés, par exemple).
- Les types intervalles font bien entendu partie des types scalaires (puisque, derrière un type intervalle, il y a toujours un type scalaire qui en permet la définition). On retrouve d'ailleurs bien dans la définition même d'un type intervalle cette caractéristique des types scalaires : si on donne deux constantes d'un même type scalaire, il est possible de recréer l'ensemble des constantes présentes entre les deux (grâce à l'existence de la fonction succ).
- Il est évidemment impossible de définir un type intervalle au départ des types réels (et ses sous-types) et chaînes de caractères. En effet, la fonction succ ("au suivant") n'a pas de sens pour les données de type real ou string. Que serait le successeur de 1.8 ou de 'albert' ?
- La syntaxe de la définition d'un type intervalle est la suivante :

  ```
  type identificateur du type = 1er élément de l'intervalle .. dernier
       élément de l'intervalle
  ```

  on notera le symbole = et les deux points (et non trois points, comme on pourrait s'y attendre)

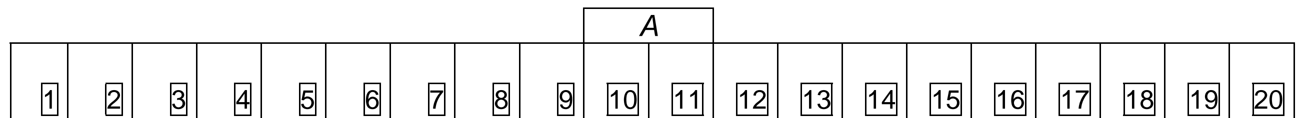
4. Les tableaux

Et la démonstration (tupperware)se poursuit :

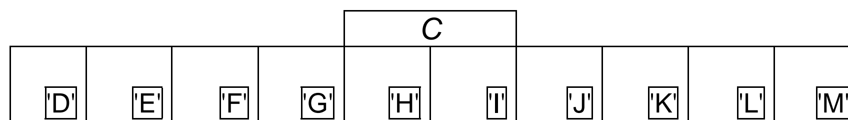
4.1 Le concept

4.1.1 Étagères et armoires

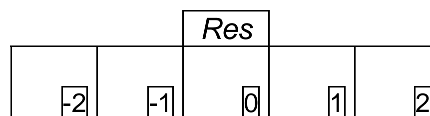
Nous avons donc la possibilité de déclarer (puis de faire utiliser par l'exécutant-ordinateur), à côté des variables "simples", des étagères comme



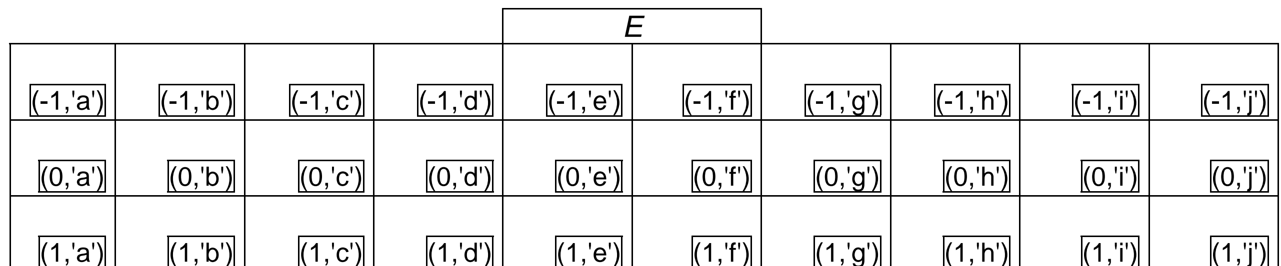
ou



ou



ou encore des "armoires" comme



Ces étagères ou armoires sont essentiellement caractérisées par les éléments suivants :

- tous les tiroirs d'une étagère ou d'une armoire donnée sont de même type;
- l'étagère ou l'armoire toute entière est désignée par un nom unique;
- les tiroirs d'une étagère sont étiquetés par des constantes scalaires (entiers, caractères, booléens, intervalles); pour définir ces étiquettes, on se contentera de préciser la première et la dernière; en ce qui concerne les armoires, chaque tiroir est caractérisé par une étiquette qui est un couple : précision de la rangée, précision de la colonne.

4.1.2 Des exemples de déclarations

On trouvera des déclarations comme les suivantes :

```
var   A,B : array[1..20] of integer;
```

définissant deux étagères comportant chacune 20 tiroirs, étiquetés de 1 à 20, destinés à recevoir des entiers (*A* est représenté ci-dessus)

```
      C : array['D'..'M'] of string;
```

définissant une étagère comportant 10 tiroirs, étiquetés de 'D' à 'M', destinés à recevoir des chaînes de caractères (*C* est représenté ci-dessus)

```
        Res : array[-2..2] of char;
```
définissant une étagère comportant 5 tiroirs, étiquetés de -2 à 2, destinés à recevoir des caractères (*Res* est représenté ci-dessus)

```
        E, F : array[-1..1 , 'a'..'j'] of real;
```
définissant deux armoires comportant chacune 3 rangées de 10 tiroirs, les étiquettes des rangées sont les entiers de -1 à 1, celles des colonnes, des caractères de 'a' à 'j'; ces tiroirs sont destinés à recevoir des réels (*E* est représentée ci-dessus).

Plutôt que de définir directement des variables-étagères ou des variables-armoires, on peut passer par la définition de types, comme ci-après :

```
type    Nombre = 1..20;
```
définissant un type intervalle parmi les entiers, constitué des entiers compris entre 1 et 20

```
        Initiales = 'D'..'M';
```
définissant un type intervalle parmi les caractères, constitué des caractères compris entre 'D' et 'M'

```
        Etagere = array[Nombre] of integer;
```
définissant un **type** d'étagère comportant 20 tiroirs, étiquetés de 1 à 20, destinés à recevoir des entiers

```
        Meuble = array[Initiales] of string;
```
définissant un **type** d'étagère comportant 10 tiroirs, étiquetés de 'D' à 'M', destinés à recevoir des chaînes de caractères

```
        Dressoir : array[-2..2] of char;
```
définissant un **type** d'étagère comportant 5 tiroirs, étiquetés de -2 à 2, destinés à recevoir des caractères

```
        Lettre = 'a'..'j';
```
définissant un **type** intervalle parmi les caractères, constitué des caractères compris entre 'a' et 'j'

```
        Buffet : array[-1..1 ,Lettre] of real;
```
définissant un **type** d'armoire comportant chacune 3 rangées de 10 tiroirs, les étiquettes des rangées sont les entiers de -1 à 1, celles des colonnes, des caractères de 'a' à 'j'; ces tiroirs sont destinés à recevoir des réels

puis

```
var     A,B : Etagere;
```
définissant deux étagères de type Etagere (*A* est représenté ci-dessus)

```
        C : Meuble;
```
définissant une étagère de type Meuble (*C* est représenté ci-dessus)

```
        Res : Dressoir;
```
définissant une étagère de type Dressoir (*Res* est représenté ci-dessus)

```
        E, F : Buffet;
```
définissant deux armoires de type Buffet (*E* est représentée ci-dessus).

- Comme montré ci-dessus, il est donc possible de définir des **types d'étagères ou d'armoires**; il faut bien saisir qu'en définissant ces types (comme Etagere, Meuble ou Dressoir) on a seulement, en quelque sorte, tracé les **plans** d'étagères ou armoires à venir,

mais qu'on n'a encore aucune variable de ces types. On notera à nouveau le symbole = utilisé pour la définition d'un type.

- Lorsque des types intervalles sont explicitement définis, on peut en user dans la définition de l'intervalle des étiquettes d'une étagère ou d'une armoire (comme Nombre, Initiales ou Lettre).

4.1.3 Un peu de vocabulaire

On ne parle évidemment pas d'étagère, d'armoire, d'étiquettes et de tiroirs dans les traités de programmation ou les manuels de Pascal. Les termes utilisés sont plutôt les suivants :

- on parle de **tableau** ou de variable indicée, plutôt que d'étagère (ou d'armoire);
- on parle de **composante** d'un tableau plutôt que de tiroir;
- on parle **d'indice** plutôt que d'étiquette.

Ainsi, on dira "la **composante** d'**indice** 3 du **tableau** *A*" plutôt que "le **tiroir** d'**étiquette** 3 de l'**étagère** *A*".

Je me permettrai dans la suite d'employer assez indifféremment des termes qui sont pour moi synonymes : étagère et tableau, tiroir et composante, étiquette et indice. Les termes imagés ont l'intérêt d'être justement porteur de métaphores, les termes consacrés ont l'avantage d'être plus répandus et plus classiques.

4.2 Les facettes essentielles du concept de tableau

4.2.1 Les composantes d'un tableau donné sont toutes de même type

Toutes les composantes d'un tableau donné doivent être de même type. Ce type peut être l'un quelconque des types déjà connus ou des types que nous découvrirons dans la suite.

On peut même avoir des tableaux de tableaux; ainsi, par exemple :

```
var   Statistiques : array[1..31] of array[1..24] of real;
```

qui pourrait d'ailleurs être rendu par

```
var   Statistiques : array[1..31 , 1..24] of real;
```

Mais il est donc impossible au sein d'un tableau donné d'avoir un mélange de tiroirs de types divers.

4.2.2 La grandeur d'un tableau donné est fixe et non modifiable au cours de l'exécution

Dès la déclaration d'un tableau, le nombre de ses composantes est figé par le choix qui est fait des étiquettes utilisées. Ainsi, en reprenant les exemples ci-dessus : *A* comporte 20 composantes, *Res* en comporte 5. Ce nombre de composantes ne peut varier au cours du programme.

Ce nombre de tiroirs ne pourra jamais être modifié en cours d'exécution; s'il ne convient pas, c'est le texte même du programme qui doit être modifié. Les tableaux en Pascal ne sont donc pas dynamiques ou à bornes variables.

Un tableau peut avoir une dimension (étagère, comme *A*, *B*, *C* et *Res*); dans ce cas chaque tiroir est repéré par une seule étiquette (d'un type scalaire). Un tableau peut aussi être à deux dimensions (armoire, comme *E* et *F*); dans ce cas chaque tiroir est repéré par un couple d'étiquettes : celle précisant la rangée et celle précisant la colonne. On peut également avoir des tableaux à trois dimensions; dans ce cas chaque tiroir est repéré par un triplet d'étiquettes : celle précisant la rangée,

celle précisant la colonne et celle précisant la tranche. On peut même avoir des tableaux de dimension 4 et au-delà.

4.2.3 Les bornes du tableau doivent être des constantes

Les étiquettes intervenant entre crochets dans la définition d'un tableau (ou d'un type de tableau) doivent être des constantes. Pas question donc que la première ou la dernière étiquette d'un tableau soit une variable (qui serait par exemple précisée ensuite par lecture) : les bornes d'un tableau sont des constantes fixées dans la partie déclaration du programme.

Les étiquettes doivent être d'un type scalaire. Il est donc impossible qu'un tableau ait des étiquettes de type réel ou string.

4.3 L'utilisation des tableaux

4.3.1 La manière de désigner une composante

Pour préciser de quel tiroir d'un tableau il s'agit, on cite le nom du tableau suivi, entre crochets, de la valeur de l'étiquette correspondante (ou des étiquettes dans le cas ou le tableau est de dimension 2 ou au delà). Cette étiquette peut être écrite comme une constante, une variable ou une expression, pourvu qu'elle ait le type scalaire attendu.

Ainsi, en se référant aux exemples donnés plus haut, on pourra écrire :

> `A[15]`, ou
>
> `A[Compteur]` pour autant que *Compteur* soit une variable entière et que sa valeur soit comprise entre 1 et 20, ou
>
> `A[succ(Compteur mod 20)]`, ou
>
> `C['E']`, ou
>
> `E[Compteur, succ(Extrait)]` pour autant que *Compteur* soit une variable entière et que sa valeur soit comprise entre -1 et 1 et que *Extrait* soit une variable de type char dont la valeur est comprise entre '`' et 'i' (pour que succ(*Extrait*) soit entre 'a' et 'j').

On notera cependant qu'il est indispensable d'utiliser une variable tableau et non un type de tableau. Ainsi, les écritures suivantes

> `Etagere[...]`, ou
>
> `Meuble[...]`, ou
>
> `Dressoir[...]`

sont incorrectes puisqu'il s'agit de types de tableau et non de tableaux..

Il faut évidemment veiller à n'utiliser que des composantes d'indices valides. Ainsi

> `A[Compteur]` n'a pas de sens si *Compteur* vaut 0 ou est strictement plus grand que 20,
>
> `C['A']` n'a pas de sens, puisque les indices de *C* vont de 'D' à 'M'.

La référence à un tiroir inexistant d'un tableau conduira donc à une erreur lors de l'exécution.

4.3.2 Les opérations permises

La seule manipulation globale permise sur les tableaux en tant qu'entité est l'affectation entre deux tableaux de même type (mêmes étiquettes, tiroirs de même type).

Ainsi, on pourra écrire (en se référant aux exemples de la page 41) :

`A := B`: les contenus de tous les tiroirs de *B* prennent place dans les tiroirs correspondants de *A*

Il est par exemple impossible de remplir par lecture ou d'afficher d'un coup tout un tableau. Ainsi

`readln(A)` ou `writeln (B)` n'ont pas de sens.

Pour remplir par lecture A, il faut écrire une boucle au sein de laquelle on effectuera la succession des lectures nécessaires pour remplir l'un après l'autre les tiroirs de A. Pour afficher les contenus des tiroirs de B, il faut faire de même : on ne peut afficher que tiroir par tiroir.

5. Enfin un autre exemple : autour de jets de dés

Il va s'agir de faire lancer 1000 fois un dé par l'ordinateur, puis d'obtenir la fréquence de chaque résultat.

5.1 Quoi faire ,

Voici la succession des affichages souhaités :

```
Je vais lancer pour vous 1000 fois un dé
Je vous donnerai ensuite la fréquence de chacun des
résultats.
Frappez Entree pour commencer
```

puis

```
11622521331316126352565212325231625636511141415455
42546462521536446531565521654652142612462122432145
51111553435335511112524236224562325545344651331145
16656255341263162116451512642363331652631632541464
23225645415126536112133313435542346425463345325656
25225362126265623612143115234212152544612651113311
45426131342543161111561256314564465566551321526616
63533553446161434454633622355145324165631246136665
62145215433156145233455214213214632365315213663233
43455344223542612634653416312453131245612523216514
14452654355236465241333465563132623144422364433565
44621636251316356534262331612114251533416262563425
33363453223333126436543264246241356265526233541323
42443455163544456446316156246511362646531145345621
16131331322126234123153154351124542633155324316461
55635561245533263164413151213434253156253563151446
45246221312621631236311423466255513263221661242444
35252166126513335636416436644311646235311516241446
62122665142442251561363351562165126365611252315146
55435432312544154224246342452342443145523655316433
Tapez Entree pour poursuivre
```

comme on le voit, l'affichage se fait à raison de 50 résultats par ligne, sur 20 lignes en tout; et enfin :

```
Voici les fréquences :
La face 1 est sortie 175 fois
La face 2 est sortie 163 fois
La face 3 est sortie 176 fois
La face 4 est sortie 154 fois
La face 5 est sortie 170 fois
La face 6 est sortie 162 fois
```

5.2 Comment faire

Face à ce problème et si nous disposons d'un dé, la démarche sera la suivante :

- On commence par aligner sur une feuille de papier les nombres de 1 à 6 :

  1 2 3 4 5 6

- A 1000 reprises
  - on lance un dé, on annonce le résultat
  - et on ajoute un petit trait en dessous du numéro sorti, par exemple

  1 2 3 4 5 6

  I (sous le 3)

- On passe ensuite en revue les traits notés sous les 6 nombres pour annoncer la fréquence des divers résultats.

5.3 Comment faire faire

5.3.1 Les variables nécessaires

On peut aisément imaginer :

- que les nombres de 1 à 6 joueront un rôle essentiel (en Pascal, on définira donc un type intervalle Face constitué des entiers entre 1 et 6; (`type Face = 1..6`);
- qu'un tableau sera nécessaire

Fréquence					
1	2	3	4	5	6

il comportera 6 tiroirs pouvant contenir des entiers, portant comme étiquettes les entiers de 1 à 6; en Pascal, on écrira donc

```
Frequence : array[Face] of integer;
```

- qu'il nous faudra :
  - une variable *Compteur* (pour compter les lancers, puis permettre une boucle d'affichage des résultats); elle sera de type entier;
  - une variable *Lancer*, de type Face (entiers entre 1 et 6), qui contiendra successivement les résultats des divers lancers.

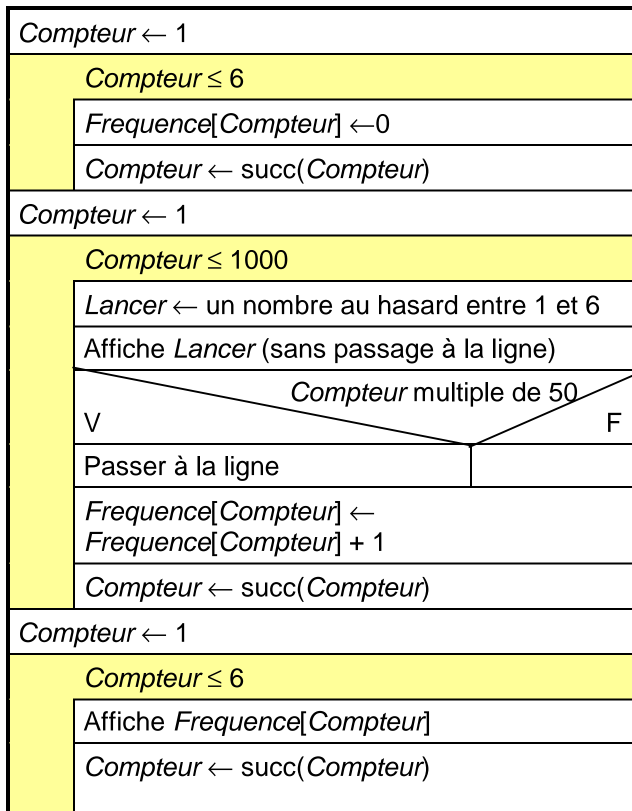
5.3.2 Attention !

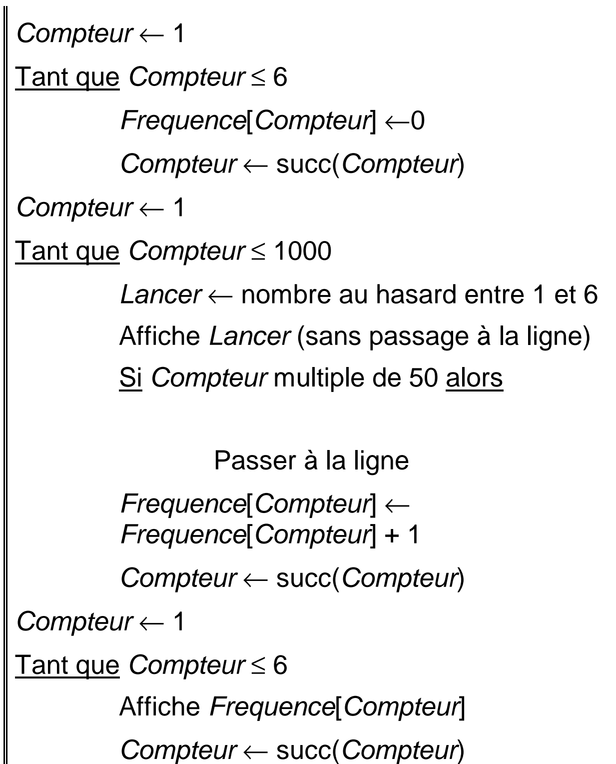
Il ne faut pas perdre de vue deux choses importantes :

- le tableau *Frequence* devra être entièrement initialisé à 0 avant le début des lancers;

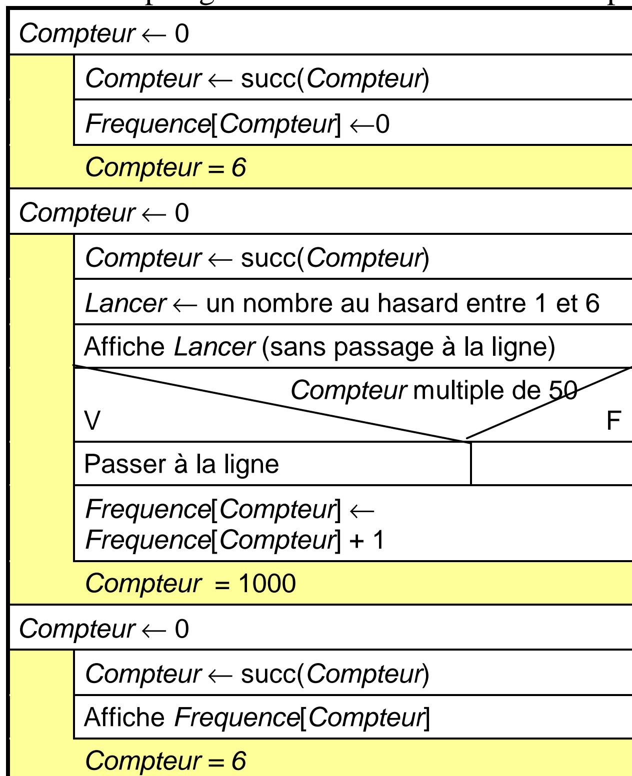
- les affichages des lancers se feront sans passage à la ligne (par write, en Pascal), mais il faudra un passage à la ligne tous les 50 lancers (par writeln).

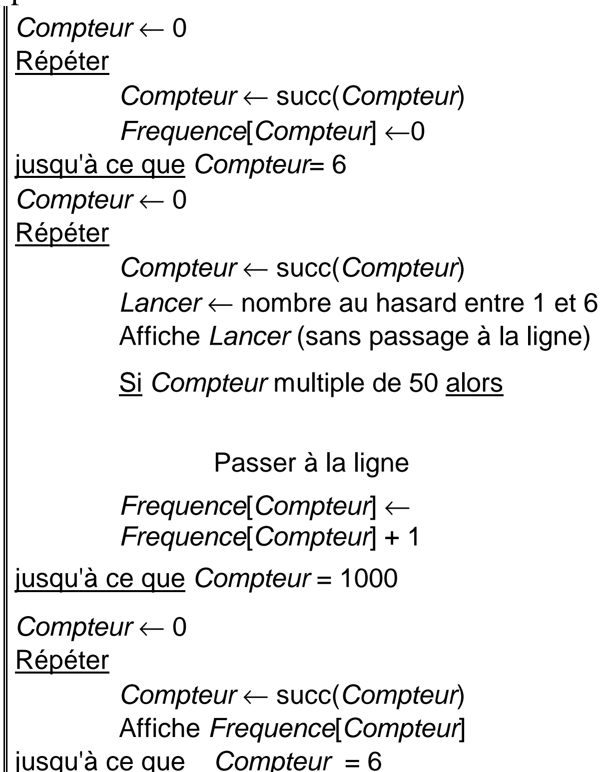
5.3.3 Marche à suivre





On aurait pu également utiliser la structure Répeter... pour décrire les boucles commandées :





On verra pourquoi nous avons retenu l'expression par la structure Tant que

5.4 Comment dire ?

En proposant directement une version habillée du programme :

```
program FREQUENCE_DE;
(* il fait simuler 1000 jets d'un dé et fait ensuite afficher la fréquence de chaque résultat *)
uses WinCrt;
type Face = 1..6; (* intervalle parmi les entiers *)                                    (1)
var   Frequence : array[Face] of integer;                                               (2)
      (* le ième tiroir contiendra le nombre de fois que la face i est sortie *)
      Lancer : Face;
      (* qui contiendra successivement les résultats obtenus lors des lancers *)
      Compteur : integer;
      (* pour initialiser le tableau Frequence, pour compter les lancers, puis pour permettre
      l'affichage des fréquences *)

begin
ClrScr; (* effacement de l'écran *)
writeln('Je vais lancer pour vous 1000 fois un dé' );
writeln('Je vous donnerai ensuite la fréquence de chacun des');
writeln('résultats.');
writeln('Frappez Entree pour commencer');
readln;                                                                                 (3)
ClrScr;
(* initialisation du tableau Frequence *)
Compteur :=1;
while Compteur <= 6 do
      begin
      Frequence[Compteur]:=0;
      Compteur := succ(Compteur);
      end;
(* lancers successifs *)
Compteur :=1;
while Compteur <= 1000 do
      begin
      Lancer:=random(6)+1;
      write(Lancer); (* Affichage sans passage à la ligne *)
      if Compteur mod 50 = 0 then                                                       (4)
            writeln; (* on passe à la ligne tous les 50 jets *)                         (5)
      Frequence[Lancer]:=Frequence[Lancer]+1;
      Compteur := succ(Compteur);
      end;
writeln('Tapez Entree pour poursuivre ');
readln;
ClrScr;
(* Affichage des fréquences *)
writeln('Voici les fréquences : ');
Compteur := 1;
while Compteur <= 6 do
      begin
      writeln('La face ',Compteur,' est sortie ',Frequence[Compteur],' fois');
      Compteur := succ(Compteur);
      end;
end.
```

(1) On définit un type intervalle Face constitué des entiers entre 1 et 6; la variable *Lancer* sera ensuite définie de type Face.

(2) On définit un tableau *Frequence* avec des indices dans l'intervalle Face et des composantes entières.

(3) L'instruction readln provoque un arrêt dans l'exécution du programme pour effectuer la lecture du caractère Entrée; dès pression de la touche Entrée par l'utilisateur, le déroulement du programme reprend. C'est souvent de cette façon qu'on interrompt des affichages pour permettre à l'utilisateur de les lire.

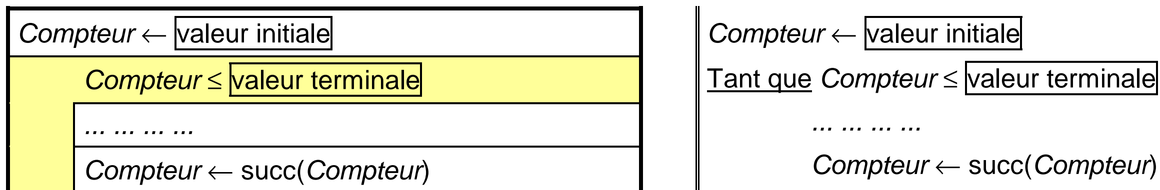
(4) On teste le fait que *Compteur* soit multiple de 50 en écrivant que le reste de la division de *Compteur* par 50 est nul, donc *Compteur* mod 50 = 0.

(5) Rappelons que writeln utilisé seul fait sauter à la ligne à l'écran.

6. La boucle Pour...

6.1 Pourquoi ?

A plusieurs reprises (trois fois dans le programme ci-dessus, et également à la page 28) nous avons eu à écrire une boucle répétitive ayant à chaque fois la même structure. Ces boucles apparaissent chaque fois sur un fond coloré et se présentent comme suit :



avec *Compteur* qui est une variable de type scalaire

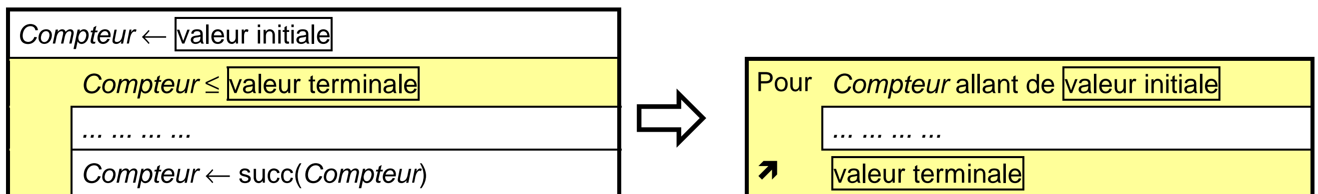
Ce qui se traduit en Pascal par

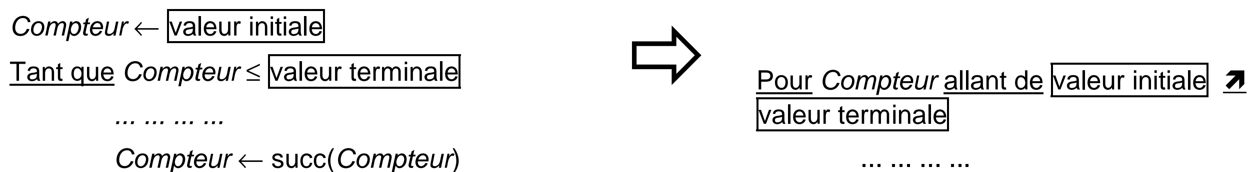
```
Compteur ← valeur initiale;
while Compteur ≤ valeur terminale do
      begin
      ... ... ... ...;
      Compteur ← succ(Compteur);
      end;
```

Ce type de répétition, entièrement gouvernée par le passage d'un compteur par une série de valeurs successives, commençant avec une valeur initiale et s'arrêtant après une valeur terminale, est très fréquent.

Dans ce type de boucle répétitive, on sait en quelque sorte d'avance combien de fois la boucle sera parcourue, puisque elle sera faite un nombre de fois correspondant au passage d'un compteur scalaire d'une valeur à une autre.

Ce type de répétition est si fréquent qu'il existe une structure qui lui correspond : la structure Pour... (For... en Pascal) :





```
Compteur ← valeur initiale;
while Compteur <= valeur terminale do
      begin
      ... ... ... ...;
      Compteur ← succ(Compteur);
      end;
```

⇨

```
for Compteur := valeur initiale to
      valeur terminale do
      begin
      ... ... ... ...;

      end;
```

6.2 La boucle Pour... "descendante"

Dans les expressions présentées ci-dessus, le compteur parcourt tout un intervalle de manière ascendante (on passe d'une valeur à la suivante, de *Compteur* à succ(*Compteur*)).

Il est bien entendu possible d'écrire des boucles où le compteur parcourt tout un intervalle de manière descendante (on passe d'une valeur à la précédente, de *Compteur* à pred(*Compteur*)).

Compteur ← valeur initiale

Compteur ≥ valeur terminale

… … … …

Compteur ← pred(*Compteur*)

⇨

Pour *Compteur* allant de valeur initiale

… … … …

↘ valeur terminale

Compteur ← valeur initiale

Tant que *Compteur* ≥ valeur terminale

… … … …

Compteur ← pred(*Compteur*)

⇨

Pour *Compteur* allant de valeur initiale ↘

valeur terminale

… … … …

```
Compteur ← valeur initiale;
while Compteur >= valeur terminale do
      begin
      ... ... ... ...;
      Compteur ← pred(Compteur);
      end;
```

⇨

```
for Compteur := valeur initiale downto
      valeur terminale do
      begin
      ... ... ... ...;
      end;
```

6.3 Syntaxe Pascal

La syntaxe de cette structure répétitive est la suivante :

```
for C := vi to vt do
      instruction unique
```

ou

```
for C := vi downto vt do
      instruction unique
```

où

- *C* est une variable de type scalaire;
- vi et vt sont deux valeurs du même type scalaire; elles peuvent être décrites comme des constantes, des variables ou des expressions;
- une instruction unique est attendue; s'il y en a plusieurs, on les enclos entre les mots begin et end.

6.4 Commentaires

- Il faut se garder de continuer à incrémenter (décrémenter) explicitement la variable compteur à l'intérieur d'une boucle Pour.... Cette incrémentation (décrémentation) est automatique. Si on ajoute une telle instruction, la variable parcourt alors l'intervalle par pas de deux.
- Le compteur d'une boucle For... ne peut donc être de type réel ou chaîne de caractères : il doit être de type scalaire (entier, caractère, booléen, intervalle ou encore *énuméré* que nous découvrirons dans la suite).
- Comme le montrent clairement les définitions et les exemples choisis, la boucle Pour... est synonyme d'une boucle Tant que....
  Ainsi, par exemple :

```
for C := 1 to 0 do
      ... ...
```

signifie

```
C:=1;
while C <= 0 do
      ... ...
```

  Donc, dans ce cas (la valeur initiale dépasse a priori la valeur terminale), le corps de la boucle **n'est pas exécuté une seule fois**.

  On a évidemment la situation duale avec la boucle For... accompagnée du mot downto.
- A la sortie d'une boucle Pour..., la valeur du compteur de boucle est indéterminée.

6.5 Retour sur les textes des programmes

Nous sommes bien entendu en mesure de réécrire en utilisant la boucle Pour... les morceaux de programme mis en évidence :

Page 28

```
CC:=1;
while CC <= Compteur do
    begin
    writeln('Donnée ',CC,' :
    ',Donnees[CC]:6:1,'; Pourcentage : ',
     Donnees[CC]/Moyenne*100:3:0);
    CC:=succ(CC);
    end;
```

⇨

```
for CC := 1 to Compteur do

    writeln('Donnée ',CC,' :
    ',Donnees[CC]:6:1,'; Pourcentage : ',
     Donnees[CC]/Moyenne*100:3:0);
```

Page 48

```
Compteur :=1;
while Compteur <= 6 do
    begin
    Frequence[Compteur]:=0;
    Compteur := succ(Compteur);
    end;
```

⇨

```
for Compteur := 1 to 6 do

      Frequence[Compteur]:=0;
```

On notera que comme la boucle Pour... ne porte que sur une seule instruction dans ces deux exemples, les mentions begin et end ne sont plus nécessaires.

Page 48

```
Compteur :=1;
while Compteur <= 1000 do
   begin
   Lancer:=random(6)+1;
   write(Lancer);
   if Compteur mod 50 = 0 then
       writeln;
   Frequence[Lancer]:=Frequence[Lancer]+1;
   Compteur := succ(Compteur);
   end;
```

⇨

```
for Compteur := 1 to 1000 do
   begin
   Lancer:=random(6)+1;
   write(Lancer);
   if Compteur mod 50 = 0 then
       writeln;
   Frequence[Lancer]:=Frequence[Lancer]+1;
   Compteur := succ(Compteur);
   end;
```

et

```
Compteur :=1;
while Compteur <= 6 do
      begin
      Frequence[Compteur]:=0;
      Compteur := succ(Compteur);
      end;
```

```
for Compteur := 1 to 6 do

      Frequence[Compteur]:=0;
```

6.6 Exercices

La fonction Factorielle est définie de la manière suivante (au choix) :

Factorielle(n) est généralement notée n! : n! = 1 x 2 x 3 x... x n pour n entier et n ≥ 1 0! = 1 n! n'existe pas si n < 0	Factorielle(n) est généralement notée n! : n! = n x (n-1)! pour n entier et n ≥ 1 0! = 1 n! n'existe pas si n < 0

Dans les éléments de programmes qui suivent,

- *N* est une variable entière et contient un entier ≥ 0
- *C* est une variable entière (compteur)

F est une variable réelle

A la fin de chacun des petits programmes suivant *F* devrait contenir *N*!. Est-ce bien le cas ? Pourquoi ?

F ← 1 Pour *C* allant de 2 ➚ *N* *F* ← *F* * *C*	*F* ← 1 Pour *C* allant de 1 ➚ *N* *F* ← *F* * *C*	*F* ← 1 Pour *C* allant de N ➘ 2 *F* ← *F* * *C*
F ← *N* Pour *C* allant de *N*-1 ➘ 1 *F* ← *F* * *C*	F ← N Pour C allant de *N*-1 ➘ 2 *F* ← *F* * *C*	

7. Un dernier exemple : la fréquence des lettres d'un texte

Il va s'agir ici de fournir la fréquence des diverses lettres minuscules présentes dans un texte fourni par l'utilisateur.

7.1 Quoi faire

Voici, comme d'habitude des exemples des affichages souhaités :

```
Vous allez me fournir un texte (sans possibilité de corriger.
les fautes de frappe éventuelles) et en marquant la fin par la
frappe du caractère $.
Je vous donnerai alors la fréquence de toutes les minuscules
non accentuées que vous aurez frappées au sein de votre texte.
Tout le travail pourra être repris à votre demande avec un texte
différent.

Donnez le texte :
Comment allez-vous? Bien j'espère!
```

et à la frappe du caractère $:

```
FREQUENCE DES MINUSCULES DANS LE TEXTE
======================================
a est 1 fois dans le texte
e est 5 fois dans le texte
i est 1 fois dans le texte
j est 1 fois dans le texte
l est 2 fois dans le texte
m est 2 fois dans le texte
n est 2 fois dans le texte
o est 2 fois dans le texte
p est 1 fois dans le texte
r est 1 fois dans le texte
s est 2 fois dans le texte
t est 1 fois dans le texte
u est 1 fois dans le texte
v est 1 fois dans le texte
z est 1 fois dans le texte

Voulez-vous recommencer avec un autre texte oui ou non ? _
```

Comme d'habitude, nous allons profiter de cet exemple pour introduire une possibilité technique supplémentaire : la saisie par l'exécutant d'un seul caractère frappé par l'utilisateur au clavier (sans attendre l'appui sur la touche Entrée) et sans que le caractère ainsi lu ne soit répercuté à l'écran.

7.2 Une information supplémentaire

Nous savons déjà qu'un texte peut être fourni au clavier par l'utilisateur, puis saisi par l'exécutant au moment de l'appui sur la touche Entrée, pour être placé dans une variable de type string. C'est l'instruction readln qui permet une telle lecture.

Ce type d'instruction de lecture présente pour l'utilisateur l'avantage que, tant qu'il n'a pas frappé la touche Entrée, il peut corriger le texte en supprimant les derniers caractères frappés grâce à la touche BackSpace. Le problème tient à la taille du texte qu'on peut ainsi faire lire puisque les variables string ne peuvent contenir qu'au plus 255 caractères. Mais, en tous cas, le texte tout entier est présent dans la variable string où il est déposé à l'issue de l'instruction de lecture.

Un autre type de lecture de caractères est possible : cette fois, chaque caractère sera saisi "au vol" par l'exécutant, sans d'ailleurs que le caractère pressé par l'utilisateur ne soit répercuté à l'écran.

C'est une **fonction** de type char et sans argument qui va offrir cette possibilité : la fonction readkey qui contient le dernier caractère frappé par l'utilisateur.

Cette manière de faire présente un net désavantage pour l'utilisateur : chaque caractère frappé est saisi (et traité) et il est trop tard pour corriger ou revenir en arrière. Il faut bien voir également que si tout un texte frappé doit être globalement présent en mémoire, ce sera au programmeur de commander les traitements nécessaires pour que l'ensemble des caractères restent présents dans une variable ou un tableau ad-hoc.

Ce sera au programmeur aussi de prévoir comment la fin du texte (=la fin des frappes successives des caractères) sera signalée.

C'est cette possibilité que nous allons utiliser ici : le texte sera saisi, caractère par caractère, jusqu'à la frappe du caractère $. Les fréquences des minuscules apparues dans la succession seront ensuite affichées.

7.3 Retour au "quoi faire ?"

7.3.1 En ce qui concerne les entrées

L'utilisateur fournira donc une suite de caractères, sans possibilité de retour en arrière; chaque caractère saisi est traité immédiatement. Ces caractères peuvent être quelconques et en nombre quelconque. La fin du texte sera marquée par la frappe du caractère $ qui interrompra donc le processus des lectures successives des caractères.

Tout ce traitement pourra être repris à la demande de l'utilisateur avec un autre texte.

7.3.2 En ce qui concerne les traitements

Seules les lettres minuscules non accentuées doivent être prises en compte.

7.3.3 En ce qui concerne les sorties

On notera que seule la fréquence des minuscules présentes dans le texte est affichée.

7.4 Comment faire

Pour venir à bout de la tâche demandée, il faut d'abord bien remarquer qu'il n'est absolument pas nécessaire de garder tous les caractères fournis en mémoire : on ne retient chaque caractère que le temps de vérifier si c'est une minuscule et le cas échéant d'ajuster sa fréquence, puis c'est le caractère suivant qui prend sa place.

Mais avant de commencer, on va inscrire la succession des minuscules :

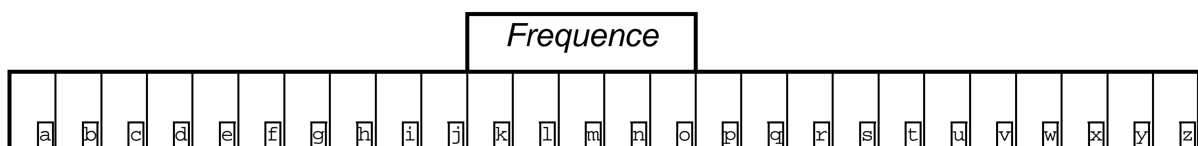
a b c d e f g h i j k l m n o p q r s t u v w x y z

pour pouvoir noter (en plaçant une barre sous la lettre correspondante) l'apparition d'une nouvelle minuscule à compter. Ensuite les actions seront les suivantes :

- on accepte un caractère, et tant qu'il ne s'agit pas de $:
- si ce caractère est une minuscule, on place un trait supplémentaire sous la lettre correspondante;
- ces lectures achevées, on passe en revue les diverses lettres en donnant le nombre de traits relatifs à chaque lettre (comportant au moins un trait);
- on reprend éventuellement le tout en n'oubliant pas, à chaque fois, de bien effacer tous les traits.

7.5 Comment faire faire

Il est évident qu'un tableau sera nécessaire :



En Pascal, il se décrira :

```
Frequence : array['a'..'z'] of integer
```

 © CeFIS - FUNDP 2002

Nous sommes en mesure de proposer une marche à suivre, en faisant apparaître au fur et à mesure, à droite, les variables nécessaires. Cette fois, nous ne la proposerons plus sous forme GNS, mais seulement en utilisant le langage de description d'algorithmes.

```
Répéter
  | Pour Compteur allant de 'a' à 'z'                                  Compteur, de type char
  |     Frequence[Compteur] ← 0
  | Caractere ← readkey                                  (1)           Caractere, de type char
  | Affiche (sans passage à la ligne) Caractere          (2)
  | Tant que Caractère ≠ '$'
  |   | Si (Caractere ≥ 'a') et (Caractere ≤ 'z') alors
  |   |   | Frequence[Caractere] ← Frequence[Caractere] + 1
  |   | Caractere ← readkey                              (1)
  |   | Affiche (sans passage à la ligne) Caractere      (2)
  | Pour Compteur allant de 'a' à 'z'
  |   | Si Frequence[Compteur] ≠ 0 alors
  |   |   | Affiche (Compteur,' est ',Frequence[Compteur],' fois dans le texte')
  | Affiche('Voulez-vous recommencer avec un autre texte oui ou non ? ')
  | Lis et place dans Reponse                                          Reponse de type string
jusqu'à ce que (Reponse='non') ou (Reponse='NON');
```

7.6 Comment dire

Et voici le programme résultant, correctement "habillé" :

```
program FREQLETTRE;

(* Ce programme fait lire un texte caractère par caractère jusqu'à la donnée d'un caractère de fin
et fait afficher pour chacune des lettres minuscules présente dans le texte sa fréquence.
Les caractères accentués ne seront pas pris en compte.
Tout le processus pourra être répété, à la demande avec un nouveau texte *)
uses WinCRT;
var Frequence : array['a'..'z'] of integer; (* pour stocker les fréquences des minuscules *)
    Caractere, (* qui contiendra successivement les divers caractères du texte *)
    Compteur (* pour accéder aux composantes successives de Frequence *)
    : char;
    Reponse (* pour savoir si l'utilisateur veut reprendre le tout *)
    : string[3];

begin
(* avertissement de l'utilisateur *)
clrscr; (* pour l'effacement de l'écran *)
writeln('Vous allez me fournir un texte (sans possibilité de corriger.');
writeln('les fautes de frappe éventuelles) et en marquant la fin par la');
writeln('frappe du caractère $.');
writeln('Je vous donnerai alors la fréquence de toutes les minuscules ');
writeln('non accentuées que vous aurez frappées au sein de votre texte.');
writeln('Tout le travail pourra être repris à votre demande avec un texte');
writeln('différent.');
writeln; (* pour passer une ligne *)

repeat
    (* initialisation du tableau Frequence *)
    for Compteur := 'a' to 'z' do
        Frequence[Compteur] := 0;
    (* Lecture du texte caractère par caractère *)
    writeln('Donnez le texte : ');
    Caractere := readkey;                                          (1)
    write(Caractere);                                              (2)
    while Caractere <>'$' do
        begin
        if (Caractere>='a') and (Caractere<='z') then
            Frequence[Caractere]:=Frequence[Caractere]+1;
        Caractere := readkey;                                      (1)
```

```
        write(Caractere);                                    (2)
        end;
    (* Affichage *)
    clrscr; (* pour l'effacement de l'écran *)
    writeln('FREQUENCE DES MINUSCULES DANS LE TEXTE');
    writeln('======================================');
    for Compteur := 'a' to 'z' do
        if Frequence[Compteur] <> 0 then
            writeln(Compteur,' est ',Frequence[Compteur],' fois dans le texte');
writeln;

write('Voulez-vous recommencer avec un autre texte oui ou non ? ');
readln(Reponse);
until (Reponse='non')or(Reponse='NON');
end.
```

(1) readkey est une **fonction** de type caractère (char) qui fournit le dernier caractère frappé au clavier; comme readkey désigne un caractère, il faut bien saisir que

readkey

employé seul ne constitue pas une instruction correcte, pas plus que

succ(*Compteur*)

ou

Caractere

Il faut indiquer ce qui doit être fait de la donnée désignée par readkey : ici on la fait placer dans la variable *Caractere*.

(2) J'ai signalé que lorsque readkey est utilisé pour saisir le caractère frappé par l'utilisateur, ce caractère n'a pas d'écho à l'écran; si donc je veux que les caractères frappés par l'utilisateur apparaissent à l'écran, à chaque fois immédiatement après qu'ils soient saisis par readkey, il me faut les faire afficher; comme chaque caractère lu est déposé dans *Caractere*, il suffit de demander l'affichage (sans passage à la ligne) de *Caractere*.

8. Exercices sur les tableaux

D'abord quelques exercices mécaniques de manipulation des tableaux, ensuite on passera à des choses plus sérieuses.

8.1 Manipulation

Pouvez vous écrire pour chacune des situations suivantes un programme Pascal qui, après avoir défini un tableau adéquat le fasse remplir comme indiqué puis (pour vérification) le fasse imprimer.

Dans chacun des cas 1 à 6, à vous de décider ce que sont les étiquettes des tableaux à définir.

1. le tableau doit comporter 20 composantes entières et être rempli de la manière suivante :

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

2. le tableau doit comporter 20 composantes entières et être rempli de la manière suivante :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

3. le tableau doit comporter 20 composantes entières et être rempli de la manière suivante :

21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40

4. le tableau doit comporter 20 composantes entières et être rempli de la manière suivante :

20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

5. le tableau doit comporter 20 composantes entières et être rempli de la manière suivante :

1	2	3	4	5	6	7	8	9	10	10	9	8	7	6	5	4	3	2	1

6. le tableau doit comporter 20 composantes de type char et être rempli de la manière suivante :

'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'i'	'j'	'k'	'l'	'm'	'n'	'o'	'p'	'q'	'r'	's'	't'

7. le tableau doit comporter 20 composantes de type char, posséder obligatoirement des étiquettes de type entier, et être rempli de la manière suivante :

'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'i'	'j'	'k'	'l'	'm'	'n'	'o'	'p'	'q'	'r'	's'	't'

8. le tableau doit comporter 10 lignes et 10 colonnes; ses composantes doivent être de type entier; il doit être rempli de la manière suivante :

1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10

9. le tableau doit comporter 10 lignes et 10 colonnes; ses composantes doivent être de type entier; il doit être rempli de la manière suivante :

1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8	8	8
9	9	9	9	9	9	9	9	9	9
10	10	10	10	10	10	10	10	10	10

10. ![3] le tableau doit comporter 10 lignes et 10 colonnes; ses composantes doivent être de type entier; il doit être rempli de la manière suivante :

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30

[3] Pour les exercices assortis du symbole !, le texte d'un programme Pascal constituant une solution possible est disponible au sein de l'ensemble des textes des programmes fournis en complément cet ouvrage

31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

11.! le tableau doit comporter 10 lignes et 10 colonnes; ses composantes doivent être de type entier; il doit être rempli de la manière suivante :

100	99	98	97	96	95	94	93	92	91
90	89	88	87	86	85	84	83	82	81
80	79	78	77	76	75	74	73	72	71
70	69	68	67	66	65	64	63	62	61
60	59	58	57	56	55	54	53	52	51
50	49	48	47	46	45	44	43	42	41
40	39	38	37	36	35	34	33	32	31
30	29	28	27	26	25	24	23	22	21
20	19	18	17	16	15	14	13	12	11
10	9	8	7	6	5	4	3	2	1

12.! le tableau doit comporter 10 lignes et 10 colonnes; ses composantes doivent être de type entier; il doit être rempli de la manière suivante :

1	2	3	4	5	6	7	8	9	10
20	19	18	17	16	15	14	13	12	11
21	22	23	24	25	26	27	28	29	30
40	39	38	37	36	35	34	33	32	31
41	42	43	44	45	46	47	48	49	50
60	59	58	57	56	55	54	53	52	51
61	62	63	64	65	66	67	68	69	70
80	79	78	77	76	75	74	73	72	71
81	82	83	84	85	86	87	88	89	90
100	99	98	97	96	95	94	93	92	91

8.2 Utilisation

Analyser les problèmes suivants; mettre en évidence les tableaux et les variables nécessaires, concevoir l'algorithme et terminer par le codage en Pascal et l'habillage du programme.

Attention, il peut arriver que la structure de tableau soit inutile pour résoudre les problèmes proposés. Il faut donc réfléchir soigneusement à ce point.

13. On souhaite faire simuler 1000 jets successifs de deux dés en s'intéressant à la différence entre les points marqués par le premier et le second dé. On demande de fournir pour chaque résultat possible la fréquence de son apparition.
14. On souhaite faire simuler 1000 jets successifs de deux dés en s'intéressant au nombre de double 1, de double 2, ... de double 6.

L'affichage prendra donc la forme suivante :

```
       Nombre de double 1 : 28
       Nombre de double 2 : 34
       ...   ...
       Nombre de double 6 : 33
```

15. On souhaite faire simuler à 10 reprises 1000 jets successifs de deux dés en s'intéressant au nombre de double 1, de double 2, ... de double 6.

    A chacune des 10 simulations on désire obtenir les résultats sous la forme suivante :

```
Simulation n° 1

       Nombre de double 1 : 28
       Nombre de double 2 : 34
       ...   ...
       Nombre de double 6 : 33
```

```
Simulation n° 2

       Nombre de double 1 : 26
       Nombre de double 2 : 29
       ...   ...
       Nombre de double 6 : 31
```

```
Simulation n° 3
       ...    ...
```

    Après ces 10 affichages, on souhaite obtenir également la moyenne sur les 10 simulations du nombre de double 1, de double 2, ... ainsi que le pourcentage correspondant pour chacun d'eux par rapport au nombre total de "doubles" obtenus. Ce dernier affichage prendra la forme

```
Au total, sur les 10 simulations :

       Moyenne du nombre de double 1 : 29.4
       Moyenne du nombre de double 2 : 28.7
       Moyenne du nombre de double 3 : 30.3
       Moyenne du nombre de double 4 : 29.9
       Moyenne du nombre de double 5 : 34.7
       Moyenne du nombre de double 6 : 31.9

       Pourcentage du double 1 dans les doubles : 16.9
       Pourcentage du double 2 dans les doubles : 15.1
       Pourcentage du double 3 dans les doubles : 16.2
       Pourcentage du double 4 dans les doubles : 15.9
       ...
```

16. On dispose de la liste des résultats de 235 étudiants, et cela successivement pour 3 épreuves. Chacun de ces résultats est un entier entre 0 et 20.

    Ces données prennent la forme d'une longue liste de 705 entiers (3*235). On a d'abord les 235 résultats de la première épreuve, puis les 235 (dans le même ordre) de la seconde, et enfin les 235 de la troisième :

```
14
12        235 résultats à la 1ère épreuve
7
13
...
...
11
11        235 résultats à la 2ème épreuve
9
...
12
10        235 résultats à la 3ème épreuve
...
```

On a trouvé un volontaire pour entrer sans erreur tous ces nombres au clavier, mais il reste à écrire le programme qui, sur cette base, fournisse la liste des numéros d'étudiants dont la cote moyenne soit plus grande ou égale à 10, assortis de cette cote moyenne. La forme en serait :

```
1            12.3
2            11.0
4            12.6
...          ...
```

17. On dispose d'une liste donnant le relevé des achats de 38 personnes. La structure en est la suivante : pour chacune des 38 personnes, on trouve le nombre de ses achats suivi des montants de chacun de ces achats (s'il y a lieu).

Par exemple :

```
4            nombre d'achats de la première personne
3456.5       montant du premier achat de la première personne
6555         montant du deuxième achat de la première personne
3567.5       montant du troisième achat de la première personne
6788         montant du quatrième achat de la première personne
0            nombre d'achats de la deuxième personne
1            nombre d'achats de la troisième personne
4565.5       montant du premier et seul achat de la troisième personne
etc.
```

On demande d'écrire un programme qui, sur base de la donnée de cette liste fasse afficher pour chacune des personnes la somme des achats et la moyenne des achats. On souhaite de plus voir afficher la moyenne globale des achats (pour toutes les personnes).

2

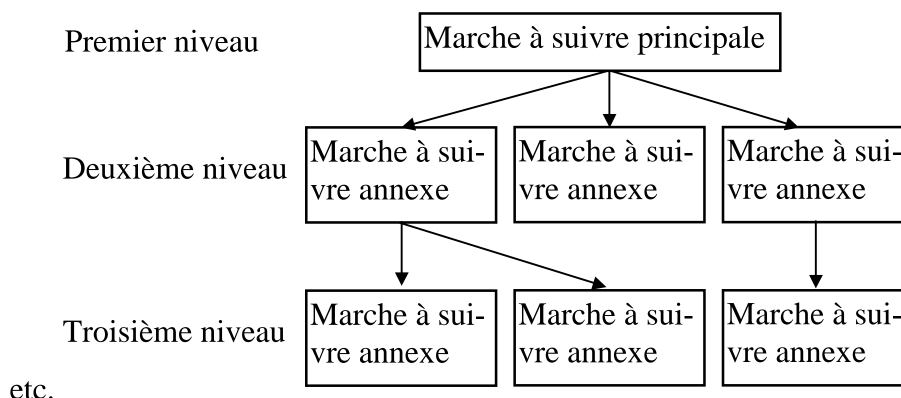
Diviser pour régner

Il est temps à présent d'en venir à l'un des points culminants de ce pays de la programmation : l'**approche descendante**[4] (ce que les anglo-saxons appellent "top down programming").

Du haut de ce "point de vue", nous allons découvrir que, même s'il n'y a pas de panacée pour gérer la complexité - et Dieu sait que la programmation peut déboucher sur du complexe - nous trouverons dans l'approche descendante une "méthode" pour découper pas à pas le traitement d'une tâche en constituants plus élémentaires, sans pour autant nous perdre dans des détails.

Ainsi, l'analyse que nous conduirons dorénavant de la tâche (à faire faire) se fera en quelque sorte par **niveaux** successifs : au tout premier niveau (celui du programme principal) nous écrirons une marche à suivre parfaitement complète mais qui mettra en évidence et laissera subsister un certain nombre **d'actions** dont nous savons pertinemment que l'exécutant-ordinateur n'est pas capable. Il nous restera donc ensuite à reprendre chacune de ces actions (qui auront été complètement décrites au premier niveau en ce qui concerne leur "Quoi faire ?") pour analyser à un **second niveau**, chacune d'elle (Comment faire ? Comment faire faire ? Comment dire ? ces actions trop complexes). Cette analyse de second niveau fera éventuellement apparaître d'autres actions encore trop complexes dont l'analyse ultérieure donnera naissance à un **troisième niveau**, etc..

La structure globale de la marche à suivre prendra dès lors la forme, non d'une longue suite d'instructions élémentaires, mais d'une arborescence dont chacun des niveaux sera consacré à l'analyse des actions complexes mises en évidence au niveau précédent :



Nous connaissons, depuis le début l'outil, essentiel qui va nous permettre de morceler l'écriture d'une marche à suivre : c'est **l'appel de procédure** qui, en autorisant pour un moment à prêter à l'exécutant des capacités qu'il n'a pas (et qu'il nous faudra donc décortiquer une par une) va

[4] Sans jeu de mots...

conduire à un style d'écriture des programmes par "blocs". Nous allons user et abuser de la phrase-miracle "fais tout ce qu'il faut pour ...".

Mais, fini de décrire ce que sera le panorama découvert au sein de cet important chapitre. Partons à l'exploration de cette colline essentielle avec, comme toujours, un exemple qui nous servira de fil d'Ariane : le tirage du Lotto[5].

1. Simulation du tirage du Lotto

Ce mot "simulation" signifie simplement que le phénomène réel du tirage du Lotto, avec ces (vraies) boules qui tournent dans un (vrai) panier et qui sont peu à peu choisies, va être "imité" par l'ordinateur.

> On parle de simulation à chaque fois qu'un phénomène "réel" (chute d'un corps, dilatation d'une barre sous l'effet de la chaleur, ...) est "représenté" (par un programme adéquat) dans l'ordinateur. La barre métallique sera montrée à l'écran, la température "virtuelle" de cette barre "virtuelle" sera "virtuellement" augmentée et l'on verra et l'on pourra mesurer (à l'écran) l'allongement "virtuel" résultant. Beaucoup de phénomènes peuvent ainsi être modélisés. Nous avons par exemple, dans le chapitre précédent, simulé des jets de dés. Finalement, c'est le comportement de l'ordinateur (gouverné par un programme qui rend compte du comportement théorique connu du phénomène) qui va singer ce que serait le phénomène réel.
>
> A l'heure où l'on parle de "réalité virtuelle", les amateurs de "discussion philosophique" sur ce qu'est le réel et ce qu'est un modèle, liront avec profit (et délectation) quelques petits chefs d'oeuvre : "Esprits, cerveaux et programmes" de John Searle ou "Conversation avec le cerveau d'Einstein" et "Le test de Turing : conversation dans un café" de Douglas Hofstadter. Ces réflexions sont reprises dans [Hofstadter 87] (voir page 231).

1.1 Description floue

Je voudrais que l'ordinateur "effectue" le tirage du Lotto.

1.2 Quoi faire (faire) ?

Je poursuivrai l'habitude prise de préciser d'abord et partiellement ce qui est souhaité, simplement en montrant les écrans successifs qui sont attendus et qui illustrent les échanges entre l'utilisateur et l'exécutant.

```
Je vais procéder pour vous au tirage des numéros du Lotto
Frappez Entrée
```

et après arrêt (jusqu'à la frappe de Entrée par l'utilisateur) puis effacement de l'écran :

5 Ou du Loto, pour tous ceux de nos lecteurs qui lisent "quatre vingt seize" le nombre 96.

```
  Voici les résultats du Lotto :

         1  41  24   5  37   2

  et le numéro complémentaire : 12


Recommence-t-on un tirage (O ou N)?
```

Comme on le voit (et comme on s'y attendait) les interventions de l'utilisateur sont ici plus que restreintes : il se contentera de préciser, à l'issue d'un tirage, s'il souhaite qu'un nouveau tirage soit effectué. Pour le reste, il lui suffira de regarder l'écran (et de prendre note des résultats, s'il pense que l'ordinateur a des chances de lui donner le prochain tirage du "vrai" Lotto).

En vrac, voici quelques unes des caractéristiques attendues :

- L'ordinateur tirera donc 6 numéros, puis un septième (le complémentaire) parmi les nombres entre 1 et 42[6]. Les 7 numéros sont bien entendus pris au hasard et **différents**.
- Les numéros tirés seront présentés en n'étant pas nécessairement triés, ni non plus dans un ordre chronologique : on garde toute liberté à cet égard.
- A l'issue de chaque tirage, l'utilisateur pourra choisir de reprendre ou d'abandonner.
- Avant le premier tirage, un court message de présentation sera affiché et disparaîtra après que l'utilisateur ait frappé la touche Entrée (cf. le premier écran montré ci-dessus).

1.3 Comment faire ?

Si nous n'avions pas à nous soucier du fait qu'il faudra dans la suite tenir compte des contraintes de l'exécutant, il serait facile d'imaginer une stratégie : je place 42 bouts de papier numérotés de 1 à 42 dans un chapeau, je mélange soigneusement puis je les tire un à un en annonçant au fur et à mesure chaque numéro ainsi obtenu, du 1er au 7ème (qui sera appelé complémentaire). Fort de cette "solution", je pourrais déjà imaginer ce qu'elle deviendra lorsque (dans la phase du "comment faire faire ?") je tenterai de l'adapter pour l'exécutant-ordinateur. J'aurai bien entendu la possibilité d'exiger de cet exécutant le tirage de numéros au hasard entre 1 et 42 (grâce par exemple en Pascal à la fonction random (Cf. p 32)).

Mais malheureusement, rien n'assure que, lors des tirages successifs, les résultats seront bien différents : l'exécutant pourra très bien obtenir sur les 7 tirages plusieurs fois les mêmes numéros. Tout se passe donc en quelque sorte comme si, avec l'ordinateur, on s'obligeait, après chaque tirage à remettre le numéro tiré dans le chapeau. C'est absurde, j'en conviens, mais je n'ai pas le choix : lors des tirages successifs de nombres au hasard l'ordinateur peut fort bien obtenir à deux (ou plusieurs) reprises les mêmes résultats[7]. Si donc je veux que les stratégies découvertes à l'étape du "comment faire ?" préparent valablement la solution imaginée à l'étape du "comment faire faire ?" (au cours de laquelle l'exécutant impose ses contraintes), il est indispensable que je transporte dans le monde de la tâche des contraintes qui viennent en réalité du monde de l'exécutant.

---

[6] Ou 49 pour ceux qui parlent du Loto.

[7] Nous avons pu le constater avec les exemples précédents qui utilisaient cette possibilité de l'exécutant de tirer des nombres au hasard.

Ce sera toujours le cas que l'étape du "comment faire ?", où théoriquement l'on est seul aux prises avec la tâche à effectuer devra anticiper sur celle du "comment faire faire ?" où l'exécutant imposera ses contraintes. Il est inutile d'imaginer des manières "humaines" d'accomplir la tâche souhaitée si les solutions envisagées se heurtent ensuite à des caractéristiques de l'exécutant qui les rendent inopérantes ou impraticables. D'ailleurs, depuis le début, nous savons que le problème de programmation naît de la confrontation de deux univers : d'une part, celui de la tâche, d'autre part, celui de l'exécutant qui sera finalement, sous notre contrôle, chargé de la mener à bien.

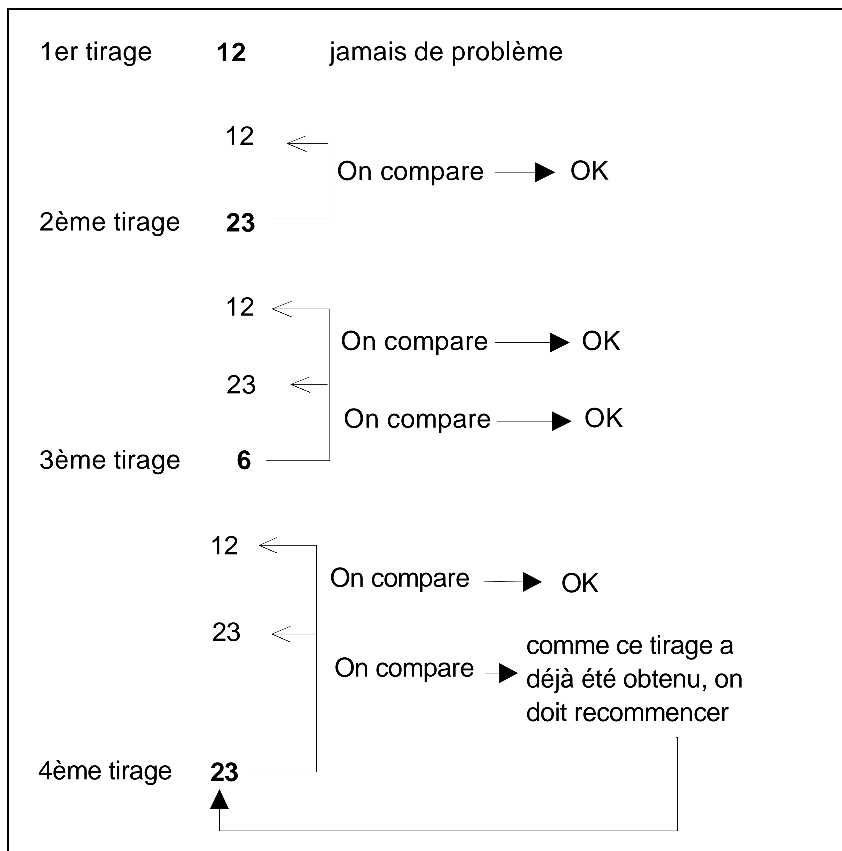
Ainsi, si je souhaite que mes réflexions face à la tâche s'adaptent ensuite en une marche à suivre qui tiendra compte de ce que m'impose l'exécutant, je dois faire comme si on m'obligeait, après chaque tirage à remettre les numéros tirés dans le chapeau, avant un nouveau tirage (où ce numéro aura donc des chances de sortir à nouveau). On le devine, il faudra que de l'une ou l'autre manière je retienne les numéros déjà obtenus, pour les comparer à tout nouveau tirage effectué.

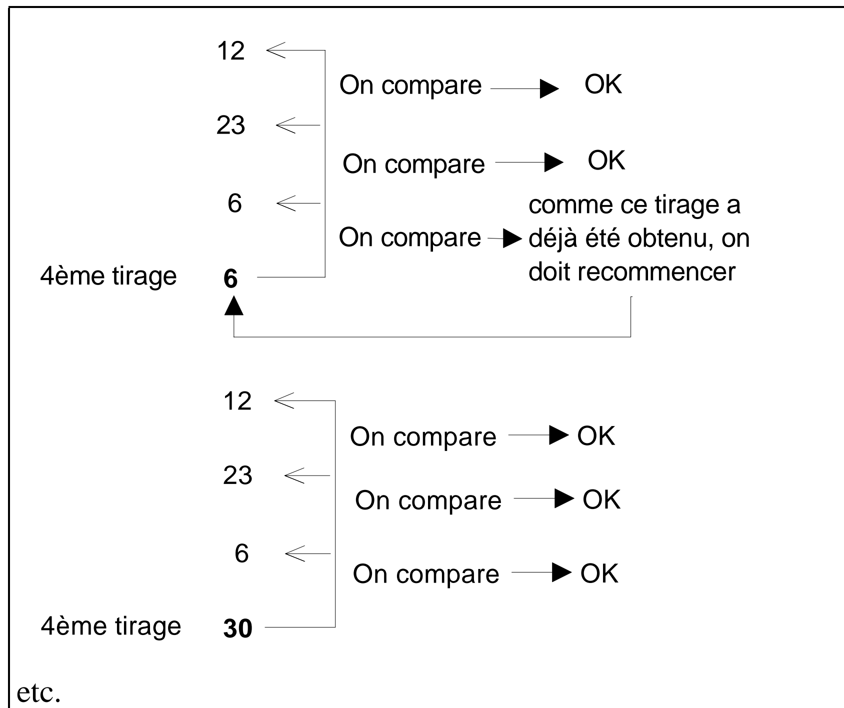
1.3.1 Première stratégie

? Comment vous y prendriez-vous pour mener à bien le tirage, si l'on vous forçait à remettre chaque numéro tiré dans le chapeau ?

Il n'y a en tout cas aucun problème pour le tout premier tirage : je tire un numéro du chapeau, je le retiens ou je le note, et je le replace dans le chapeau. Les choses se compliquent dès le second tirage : le numéro sorti du chapeau doit être comparé à celui déjà retenu. Si par malheur ils étaient identiques, il me faudrait (après avoir remis le numéro indésirable dans le chapeau) retirer à nouveau, et recommencer la comparaison, puis éventuellement le tirage, ... Ainsi donc le tirage des numéros de 2 à 7 doit à chaque fois s'accompagner d'une comparaison aux numéros déjà tirés.

Le déroulement pourrait donc être par exemple :





Ainsi donc la démarche pourrait se décrire :

- On tire le 1er numéro, on le note ... et on le replace dans le chapeau
- Du 2e au 7e numéro
  - On tire un numéro au hasard (en s'obligeant ensuite à le remettre dans le chapeau);
  - On le compare à ceux déjà notés et le cas échéant on recommence le tirage jusqu'à ce que le numéro tiré soit différent de tous ceux notés jusque là.
- On passe alors en revue les 7 numéros finalement notés en annonçant les 6 premiers retenus et notés, puis le 7e comme étant le complémentaire.

Notons au passage qu'ici les numéros seront annoncés dans l'ordre chronologique de leur tirage.

1.3.2 Deuxième stratégie

Soyons clairs : cette seconde manière de procéder n'est généralement proposée que par ceux ayant déjà une expérience de la programmation et ayant donc intégré (souvent inconsciemment) contraintes et possibilités de l'exécutant-ordinateur. Ils transforment en réalité un "comment faire faire ?" que la connaissance de la programmation leur dicte immédiatement en un "comment faire ?" (plutôt que l'inverse).

Cette stratégie ne diffère de la précédente que par la manière dont on note (et retient) les numéros déjà tirés.

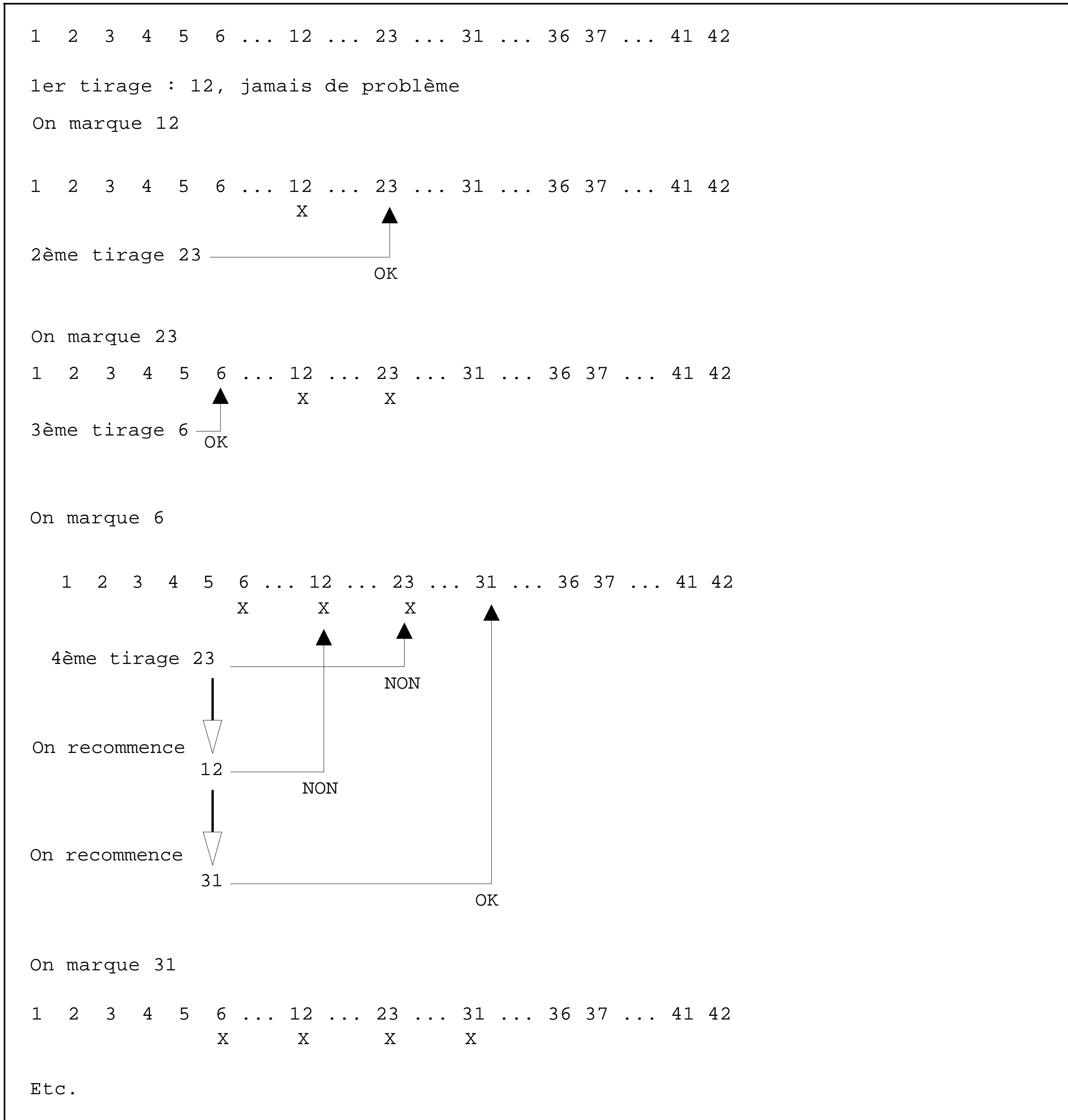
Je vais, préalablement aux tirages, écrire tous les numéros entre 1 et 42

```
1  2  3  4  5  6 ... 12 ... 23 ... 31 ... 36 37 ... 41 42
```

Le premier tirage s'effectue sans problème; je le retiens en plaçant une croix dans la liste ci-dessus sous le nombre obtenu.

A chacun des tirages suivants, je regarde si le nombre tiré n'est pas déjà repéré par une croix. Si c'est le cas, je dois recommencer, puis vérifier à nouveau, et cela jusqu'à ce que le nombre tiré ne soit pas coché. Lorsque j'obtiens finalement un nombre acceptable, je le retiens en le marquant d'une croix.

Par exemple :



On constate la facilité avec laquelle s'effectuent les vérifications. Après les tirages lorsqu'il ne s'agira plus que d'annoncer les résultats, on pourra même sans peine les annoncer en ordre croissant ("dans l'ordre arithmétique" comme on dit à la TV). Il y a malheureusement un écueil : c'est que si

j'annonce toujours le 7ème qui est marqué comme étant le complémentaire, ce complémentaire sera forcément à chaque fois plus grand que les 6 autres numéros. Et ça, ce n'est pas une simulation correcte du tirage du Lotto !

La solution est simple : du 1er au 6e tirage, je retiendrai les numéros choisis en marquant simplement d'une croix les 6 nombres concernés. Mais pour le tirage du 7ème et dernier (le complémentaire), plutôt que de marquer d'une croix le nombre correspondant à ce tirage, je le noterai (après les vérifications d'usage) à part sous mon tableau comme ci-dessous :

```
1  2  3  4  5  6 ... 12 ... 23 ... 31 ... 36 37 ... 41 42
      X        X      X      X      X         X


Complémentaire : 29
```

En résumé, cette seconde stratégie peut se synthétiser sous la forme

- On aligne les numéros de 1 à 42.
- On tire un nombre au hasard et on marque d'une croix dans la liste le numéro correspondant (puis on remet (malheureusement !) le nombre tiré dans le chapeau).
- Du 2e au 6e tirage
  - on tire un nombre au hasard (comme ci-dessus)
  - si le numéro correspondant est déjà marqué d'une croix on recommence le tirage (en replaçant bien sûr dans le chapeau) et cela jusqu'à ce que le nombre sorti ne soit pas marqué;
  - on marque d'une croix dans la liste le nombre finalement retenu.
- Pour le tirage du complémentaire
  - on tire un nombre au hasard (comme ci-dessus)
  - si le numéro correspondant est déjà marqué d'une croix on recommence le tirage et cela jusqu'à ce que le nombre sorti ne soit pas marqué;
  - on écrit sous la liste le nombre finalement retenu.
- On annonce les 6 numéros marqués de la liste puis on annonce le 7e nombre (figurant sous la liste) comme complémentaire..

Me voici donc avec 2 stratégies, qui, puisqu'elles tiennent d'emblée compte des caractéristiques de l'exécutant, vont donner naissance à deux marches à suivre bien différentes.

? Voyez-vous une autre manière de procéder pour ne pas devoir traiter complètement à part le tirage du complémentaire ?

1.4 Comment faire faire à un premier niveau d'analyse (pour la 1ère stratégie)

Il nous reste à mettre en oeuvre, dans un choix de variables et dans l'écriture d'une marche à suivre, la stratégie retenue (cela vaudrait la peine de la relire avant de démarrer).

Mais dorénavant, nous allons prendre l'habitude d'une démarche, à cette étape du "comment faire faire ?", qui se scinde en 2 préoccupations

- le choix d'une structure de données (ou de plusieurs structures s'il le faut)
- l'écriture de la marche à suivre, à un premier niveau d'analyse.

1.4.1 Structure de données.

Il s'agit, pour l'instant, d'une double interrogation :

- y a-t-il des constantes importantes inhérentes au problème traité ?
- des tableaux seront-ils indispensables ?

1.4.1.1 Constantes inhérentes au problème.

? Voyez-vous des nombres qui sont essentiels dans la description que vous feriez de ce qu'est le Lotto ?

Ce sont ceux qu'on sera bien forcé de mentionner si l'on veut décrire à quelqu'un qui ignore tout du Lotto en quoi consiste un "tirage du Lotto".

On ne pourra par exemple certainement pas passer sous silence le fait qu'on tire **7** numéros parmi les nombres de 1 à **42**.

Ainsi deux constantes 7 et 42 font en quelque sorte intimement partie de ce qu'est le Lotto. Ces constantes pourraient demain être différentes (le 42 devient d'ailleurs 49 pour nos amis français) mais je ne pourrai jamais décrire ce qu'est le Lotto sans parler du nombre de numéros tirés et du nombre de numéros possibles (pour le tirage).

Je mets donc en évidence deux constantes (deux paramètres inhérents) à la description de ce qu'est le Lotto

- NNT, le nombre de numéros tirés (actuellement synonyme de 7)
- NNP, le nombre de numéros possibles, autrement dit le nombre de boules placées dans l'urne (actuellement 42)

On pourrait ici utiliser non les abréviations NNT et NNP, mais des identifiants plus longs et plus parlant (comme NombreDeNumerosTires et NombreDeNumerosPossibles). Vous savez que je ne recule pas devant des identificateurs d'une certaine longueur. Ici, cependant, pour des facilités typographiques, je conserverai les abréviations NNT et NNP. A chacun à ce propos de faire ses choix : l'important est que les termes retenus soient significatifs et "disent" sans ambiguïté ce qu'ils sont censés désigner.

Cette réflexion préalable à propos de constantes structurellement attachées à la tâche fait partie d'une analyse complète et bien menée. Nous en reverrons maints exemples par la suite.

Dorénavant, je m'interdirai donc de parler encore de 7 et de 42 pour mentionner plutôt NNT et NNP.

1.4.1.2 Les tableaux nécessaires

? Voyez vous la nécessité d'un tableau ? Avec combien de tiroirs et quelles étiquettes ?

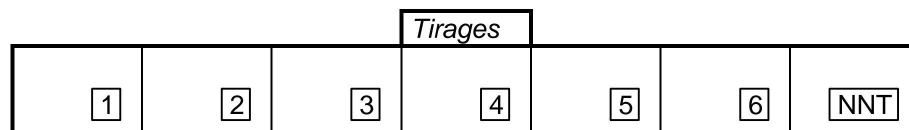
Il faudra, nous le savons, que l'exécutant "retienne" les nombres tirés au fur et à mesure que ceux-ci seront acceptés (après comparaison avec ceux déjà retenus). C'était en effet notre propre

stratégie à l'étape du "Comment faire ?". Pour faire retenir, côte à côte, ces résultats successifs, nous disposons d'une structure adéquate : celle de tableau.

Il faudra donc que l'exécutant dispose d'une étagère[8] avec autant de tiroirs que le nombre de numéros à tirer. Autrement dit nous définissons un tableau, que nous nommons *Tirages*, comportant NNT tiroirs étiquetés de 1 à NNT.

Chacun de ces tiroirs pourra contenir un entier, forcément compris entre 1 et NNP. Ces entiers entre 1 et NNP jouent un rôle fondamental dans le présent problème. On le verra, lors de l'écriture du programme Pascal, j'en ferai un type d'information spécifique, un type intervalle, celui des entiers compris entre 1 et NNP.

Peu à peu, au cours des tirages successifs, les numéros retenus viendront garnir les tiroirs prévus.



Il ne nous reste plus qu'à écrire la

1.4.2 Marche à suivre.

FAIS TOUT CE QU'IL FAUT POUR PRESENTER A L'UTILISATEUR UN ECRAN D'AVERTISSEMENT (CF LES SPÉCIFICATIONS) QUI DEVRA S'EFFACER A L'APPUI SUR LA TOUCHE ENTREE.

Répéter

Place un nombre au hasard entre 1 et NNP dans le tiroir n° 1 de l'étagère *Tirages*.

Pour *Compteur* allant de 2 à NNT — *Compteur* de type entier

FAIS TOUT CE QU'IL FAUT POUR PLACER UN NOMBRE CONVENABLE DANS LE TIROIR N° *Compteur* DE L'ETAGERE *Tirages*

FAIS TOUT CE QU'IL FAUT POUR AFFICHER LES RESULTATS CONTENUS DANS *Tirages* (CF. LES SPECIFICATIONS)

Demande à l'utilisateur s'il veut recommencer et place sa réponse dans *Reponse*. — *Reponse* de type caractère

Jusqu'à ce que *Reponse* = 'N'

Comme annoncé, nous avons eu recours à la description d'**actions complexes**, dont il nous restera, à ce premier stade de l'analyse, à préciser la teneur et les objectifs précis.

Avant de poursuivre, nous allons nous permettre de synthétiser cette première analyse en remplaçant chaque description d'instruction complexe par un verbe qui en résume l'action.

8 J'utilise indifféremment, comme au chapitre précédent, le terme imagé "étagère" ou le terme consacré "tableau".

AVERTIR

Répéter

Tirages[1] ← un nombre au hasard entre 1 et NNP

Pour *Compteur* allant de 2 à NNT

TIRER

AFFICHER

Demande à l'utilisateur s'il veut recommencer et place sa réponse dans *Reponse*.

Jusqu'à ce que *Reponse* = 'N'

A ce niveau, outre l'étagère *Tirages*, deux variables seulement sont nécessaires :

- *Compteur*, de type entier qui variera de 2 à NNT, pour baliser le tirage des numéros qui viendront remplir les tiroirs 2 à NNT de *Tirages*. Ce sera donc un entier (compris entre 2 et NNT).
- *Reponse*, qui accueillera la réponse de l'utilisateur. Nous la choisirons de type caractère (elle accueillera 'O' ou 'N').

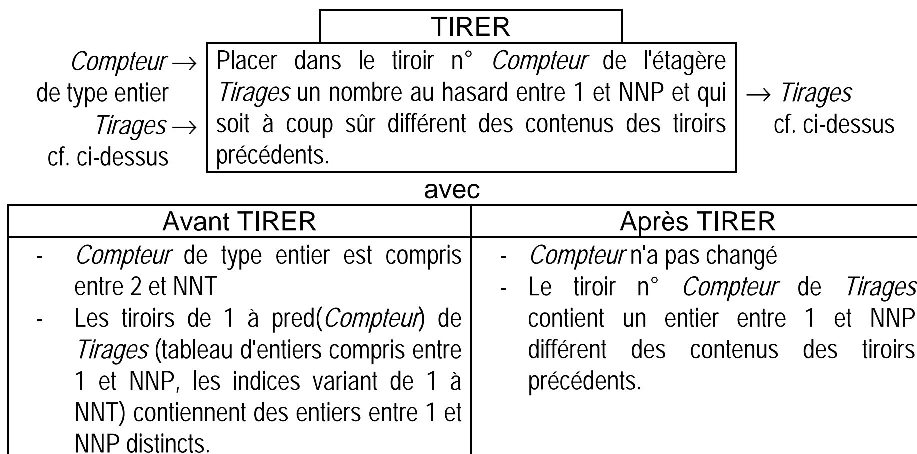
Comme on le voit, je les fais apparaître à nouveau à droite du texte de la marche à suivre.

Cette toute première marche à suivre, qui comporte encore des actions non décortiquées, correspond à ce que nous appellerons une **analyse de premier niveau**.

Cette analyse faite, il est évident que nous pourrions à présent répartir le travail restant entre plusieurs personnes (ou plusieurs équipes), chacune d'entre elles se chargeant de mener à bien l'analyse d'une des instructions complexes (AVERTIR, TIRER, AFFICHER). Chacune se concentrera sur un seul des modules, en oubliant le reste du problème et en ignorant tout du travail des autres.

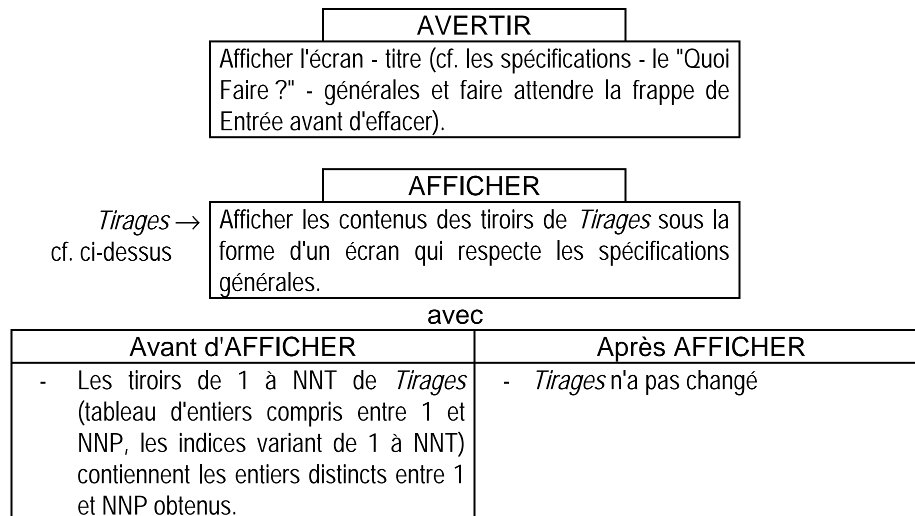
Il nous reste cependant une tâche essentielle afin qu'après cette modularisation du travail, la mise en commun débouche sur un ensemble correct et cohérent. Il nous faut, bien entendu, préciser complètement ce que signifie et recouvre chacune des actions complexes envisagées, pour qu'il ne puisse subsister aucune ambiguïté et que les problèmes correspondants soient parfaitement définis.

Autrement dit, nous allons définir le "Quoi faire ?" pour chacun des modules restant à réaliser.



Avant TIRER	Après TIRER
- *Compteur* de type entier est compris entre 2 et NNT - Les tiroirs de 1 à pred(*Compteur*) de *Tirages* (tableau d'entiers compris entre 1 et NNP, les indices variant de 1 à NNT) contiennent des entiers entre 1 et NNP distincts.	- *Compteur* n'a pas changé - Le tiroir n° *Compteur* de *Tirages* contient un entier entre 1 et NNP différent des contenus des tiroirs précédents.

 © CeFIS - FUNDP 2002



On le constate, chacune des sous-tâches est parfaitement définie : on précise ce que sera l'effet de l'action complexe décrite, ce qui est attendu.

En un mot, chacun des rectangles ci-dessus contient le "Quoi faire ?" relatif à chacune des actions complexes commandées dans la marche à suivre.

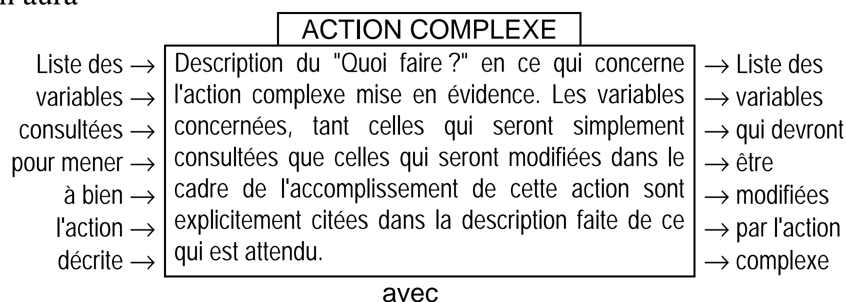
> En réalité, plutôt que de recopier complètement des morceaux des spécifications générales au sein des descriptions de AVERTIR ou de AFFICHER, on renvoit à ces spécifications. Il va de soi que dans une approche parfaitement modulaire, l'équipe responsable de l'analyse et de l'écriture de chaque module doit disposer de tout ce qui est indispensable pour mener à bien le travail de rédaction de la marche à suivre concernée.
>
> C'est le plus souvent pour des raisons d'économie de place que cette description du "Quoi faire ?" de chaque action complexe renvoit à des définitions (comme pour le tableau *Tirages*) ou des précisions (comme les spécifications de AVERTIR) données précédemment.

Ce qui fait le lien entre ces diverses questions, ce qui les intègre en quelque sorte au sein du problème commun, ce sont les variables qui seront consultées et modifiées par chacune des actions spécifiées.

Ainsi, pour TIRER, il ne s'agit pas de placer dans **un** tiroir **d'un** tableau un nombre au hasard différent des contenus des tiroirs précédents de ce tableau. Il s'agit précisément de placer dans le tiroir n° *Compteur* du tableau *Tirages* un nombre au hasard qui soit différent des contenus des tiroirs précédents.

Même si chacune des tâches précisées (AVERTIR, TIRER, AFFICHER) donnera lieu à un problème en soi (puisque le "Comment faire faire ?" reste à chaque fois à écrire), ce qui les relie ce sont ces variables - charnières, consultables ou modifiables.

En général donc on aura



Avant l'ACTION COMPLEXE	Après l'ACTION COMPLEXE
Liste des variables (consultables ou modifiables) avec précision des valeurs possibles et permises avant l'action complexe mentionnée	Liste des variables modifiables avec précision de ces modifications (et rappel éventuel des valeurs -inchangées- des variables consultables).

Plusieurs remarques importantes doivent être faites :

1. Il va de soi que rien d'autre que ce qui est précisé dans chaque rectangle décrivant le "Quoi faire ?" ne peut être évoqué concernant le problème global qui a donné naissance à ces problèmes partiels, ni non plus quoi que ce soit concernant les tâches décrites dans les autres rectangles.
2. Les seules variables préexistantes dont on puisse postuler l'existence au sein du rectangle explicatif d'une tâche sont les variables consultables (mentionnées à gauche) et les variables à modifier (mentionnées à droite). Toutes les autres variables définies lors de l'analyse du 1er niveau déjà menée doivent être considérées comme inconnues (et forcément non modifiables) au sein de la tâche partielle décrite. Ainsi, aucun des trois blocs définis plus haut ne peut postuler l'existence de la variable *Réponse,* pourtant définie dans la marche à suivre du 1er niveau.
3. Seules les variables mentionnées à droite peuvent et doivent être modifiées. Il est exclu que les variables consultables (et non modifiables) voient leur valeur changée par l'action décrite. Ainsi, *Compteur* qui doit être consulté par TIRER ne peut être modifié à l'issue de cette action (puisqu'il n'est pas mentionné à droite - à la sortie - de TIRER).
4. La connaissance des constantes définies au premier niveau est bien entendu assurée pour chacune des actions décrites; simplement, nous ne les avons pas indiquées à gauche de chacun des rectangles explicatifs afin d'alléger la présentation.

Voici venu le moment d'une petite halte qui permette d'apprécier l'importance de la démarche illustrée.

Parti d'une tâche dans son ensemble (simuler le tirage du Lotto) parfaitement précisée (à l'issue de la question "Quoi faire ?"), nous avons développé une stratégie générale qui anticipait sur les contraintes de l'exécutant ("Comment faire ?"). Enfin, nous avons précisé cette stratégie en une marche à suivre qui laissait place à des tâches trop complexes pour l'exécutant mais dont nous avons parfaitement défini la teneur et les objectifs. En un mot, nous avons précisé "Quoi faire ?" pour chacune de ces tâches partielles.

Nous voici donc revenus en quelque sorte au point de départ. Pour chacune des trois tâches définies, nous allons reprendre la démarche familière : "Comment faire ?" (TIRER ou AFFICHER ...)", "Comment le faire faire ?". Pour terminer par le "Comment dire ?" (en Pascal et sous forme de procédures).

Au travers de cette division de la tâche, de cette démarche descendante, la programmation devient en quelque sorte un perpétuel recommencement : un "comment faire faire ?" de premier niveau donne naissance à plusieurs "Quoi faire ?" qui pourront, eux-mêmes, après l'analyse qui répondra pour chacun d'eux à "Comment faire faire ?", donner naissance à de nouveaux "Quoi faire ?" ... ... etc..

1.5 Comment dire ? (l'analyse de 1er niveau de la 1ère stratégie)

Bien que nous n'ayons pas terminé le processus, nous sommes d'ores et déjà en mesure d'exprimer en Pascal le programme **principal** qui rendra compte de **l'analyse de premier niveau** menée. Nous ne pouvons évidemment pas encore compléter les textes des trois procédures qu'il reste à préciser, mais nous en rappellerons, sous forme de commentaires, les spécifications, c'est-à-dire essentiellement la teneur des 3 rectangles ci-dessus.

```
program LOTTO;
        (* Il fait simuler un tirage du lotto. On y utilise un tableau à 7 composantes pour y stocker
        les résultats obtenus *)
uses WinCRT[9];
        (* pour pouvoir utiliser les procédures de gestion d'écran comme Clrscr ou Gotoxy en Turbo
        Pascal sous Windows *)                                                                  (1)
const NNP = 42; (* le plus grand numéro pouvant être obtenu *)
      NNT = 7;  (* nombre de numéros tirés (y compris le complémentaire) *)                    (2)
type NumerosPossibles = 1..NNP;                                                                (3)

var Compteur: integer;
        (* compteur pour les boucles for *)
    Tirages : array[1..NNT] of NumerosPossibles; (* qui contiendra les numéros obtenus *)
    Reponse : char;
        (* réponse de l'utilisateur *)

procedure AVERTIR;                                                                             (4)
        (* Fait afficher l'écran - titre (cf. les spécifications générales) et attend la frappe
        d'Entrée avant d'effacer)*)

procedure TIRER;
        (* elle fait placer dans le tiroir n° Compteur de l'étagère Tirages un nombre au hasard entre
        1 et NNP et qui soit à coup sûr différent des contenus des tiroirs précédents.*)

procedure AFFICHER;
        (* elle fait afficher les contenus des tiroirs de Tirages sous la forme d'un écran qui
        respecte les spécifications générales*)

begin
randomize;                                                                                     (5)
AVERTIR;
repeat
  ClrScr; (* pour effacer avant de recommencer un tirage *)
  Tirages[1]:=random(NNP)+1;
  for Compteur := 2 to NNT do
     TIRER;
  AFFICHER;
  gotoxy(4,20);                                                                                (6)
  write(' Recommence-t-on un tirage (O ou N)?');
  readln(Reponse);
  Reponse:= upcase(Reponse);                                                                   (7)
until Reponse = 'N';
end.
```

1.5.1 Commentaires sur le programme Pascal

(1) C'est ici un détail, propre à la version envisagée de Turbo-Pascal. Ces détails sans grande importance changent avec l'implémentation choisie.

(2) C'est le mot réservé const qui annonce en Pascal la définition des constantes. Elle doit précéder les déclarations de type et de variable et associe donc un identificateur (qui respecte la syntaxe Pascal habituelle) à une valeur constante.

On peut définir des constantes entières, réelles, booléennes, chaînes de caractères, ...

9 Dans des implémentations plus anciennes de Pascal, la formule "uses WinCRT" était par exemple "uses CRT".

Attention, c'est le symbole = qui lie l'identificateur à la constante identifiée.

(3) On définit ici, puisque Pascal le permet, un type intervalle (parmi les entiers). Les entiers entre 1 et NNP seront dorénavant rebaptisés comme étant du type NumerosPossibles.

(4) On sait que c'est entre la partie déclaration et la partie exécutable que les textes des procédures (correspondant aux actions complexes) viendront se nicher.

L'ordre dans lequel les diverses procédures s'y présentent n'a rien à voir avec l'ordre dans lequel s'effectuent les appels de ces procédures au sein de la partie exécutable.

(5) L'appel de la procédure randomize est obligatoire si l'on veut que la fonction random fournisse des séries de nombres aléatoires différentes à chaque exécution. En son absence, à chaque nouvelle exécution du programme, c'est la même suite de tirages aléatoires qui est produite.

(6) J'ai d'emblée tenu compte de la présentation demandée avec l'envoi du curseur en bas de l'écran pour l'affichage de la question à l'utilisateur. C'est la procédure gotoxy(,) qui fait effectuer ce saut du curseur. (Elle fait partie dans l'implémentation Turbo Pascal 1.5 sous Windows, de la librairie WinCRT.)

(7) Un tout petit détail qui me permet de présenter un outil (de genre fonction) : upcase qui transforme les minuscules en majuscules (sauf malheureusement les minuscules accentuées qui restent ce qu'elles sont - nous y reviendrons -).

De cette manière, que la réponse de l'utilisateur soit "n" ou "N", upcase la transforme en "N".

? Aurait-on pu se passer de upcase en arrêtant cependant dès que la réponse de l'utilisateur est "n" ou "N" ?

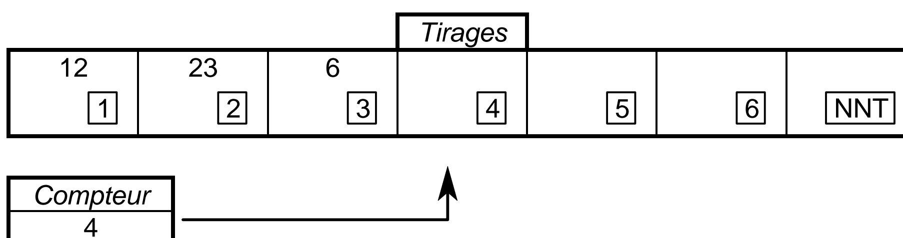
1.6 Analyse de second (puis troisième) niveaux

1.6.1 Analyse de TIRER

Nous savons parfaitement "Quoi faire ?" (C'est ce qui est explicité dans le rectangle de la page 70). Il reste à trouver d'abord "Comment faire ?".

1.6.1.1 TIRER ? Comment faire ?

Imaginons un instant la situation :



La tâche étant ici parfaitement simple (il nous faut placer un nombre convenable dans le tiroir n° 4), la découverte d'une stratégie est immédiate.

- On tire un nombre au hasard, on vérifie qu'il est bien distinct du contenu des tiroirs (de 1 à pred(*Compteur*)) et on recommence éventuellement jusqu'à ce qu'il n'y ait pas égalité (du tirage effectué avec les contenus des tiroirs précédents).

1.6.1.2 TIRER ? Comment faire faire ?

Un premier jet conduit à

<u>Répéter</u>

Place un nombre au hasard entre 1 et NNP dans le tiroir n° *Compteur* de *Tirages*

FAIS TOUT CE QU'IL FAUT POUR VERIFIER S'IL Y A EGALITE ENTRE LE NOMBRE AINSI PLACE DANS LE TIROIR *Compteur* ET L'UN DES TIROIRS PRECEDENTS.

<u>jusqu'à ce que</u> **il n'y ait pas égalité**

ou sous forme condensée

<u>Répéter</u>

Tirages[*Compteur*] ← un nombre au hasard entre 1 et NNP

VERIFIER

<u>jusqu'à ce que</u> **il n'y ait pas égalité**

Je ne puis évidemment pas me contenter de la condition "***il n'y ait pas égalité***" énoncée sous cette forme.

Il faut qu'à l'issue de l'action VERIFIER, "quelque chose" ait (éventuellement) changé et signale qu'il y a ou non égalité (entre le nombre qui vient d'être tiré et l'un des précédents). Ce "quelque chose", ce ne peut être que le contenu d'une variable, dont l'action VERIFIER aura pour but, justement, de déterminer le contenu.

Cette variable, je la choisis de type booléen et je la baptise *Egalite*, ce qui me permet de préciser la marche à suivre.

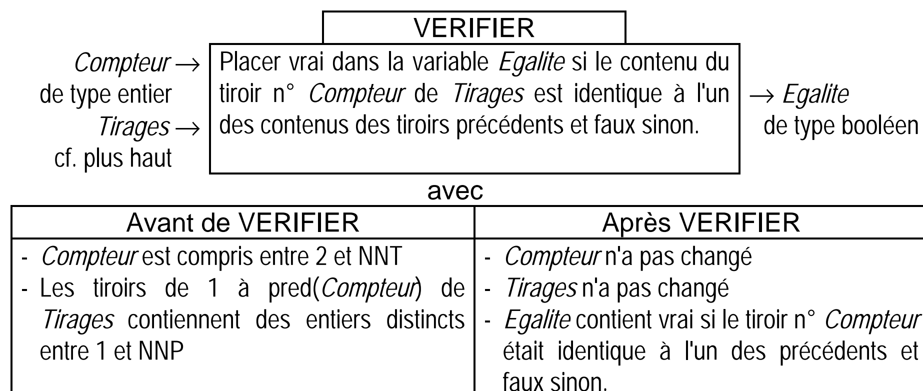
<u>Répeter</u>

Tirages[*Compteur*] ← un nombre au hasard entre 1 et NNP

VERIFIER

<u>jusqu'à ce que</u> **pas** *Egalite* *Egalité* de type booléen

Il me reste bien entendu, puisqu'une action complexe VERIFIER a été mise en évidence, à en préciser exactement la teneur :



Avant de VERIFIER	Après VERIFIER
- *Compteur* est compris entre 2 et NNT - Les tiroirs de 1 à pred(*Compteur*) de *Tirages* contiennent des entiers distincts entre 1 et NNP	- *Compteur* n'a pas changé - *Tirages* n'a pas changé - *Egalite* contient vrai si le tiroir n° *Compteur* était identique à l'un des précédents et faux sinon.

Il faut reconnaître que ce formalisme est, en ce qui concerne l'exemple traité ici, assez lourd : la description du "Quoi faire ?" est à chaque fois plus longue que la marche à suivre à laquelle elle donnera naissance!

Cependant, il vaut mieux, me semble-t-il présenter et faire apprécier (et si possible admettre) la méthodologie sur un exemple "simple" (même si sa justification n'apparaît dans ce cas pas très claire) plutôt que de le faire dans un cas où la complexité du problème traité réclame trop d'attention au détriment de la compréhension initiale de cette méthodologie.

Nous sommes dès à présent en mesure d'écrire :

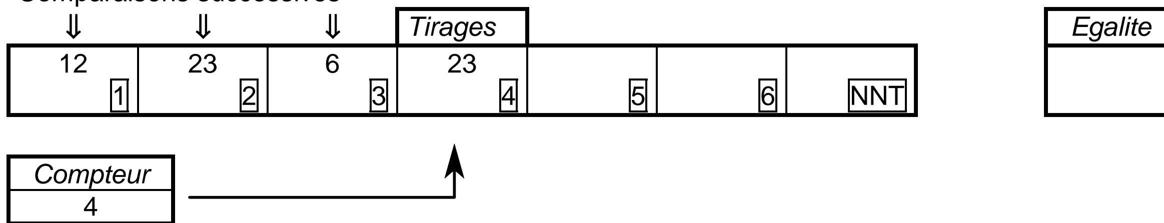
1.6.1.3 TIRER ? Comment dire ?

```
procedure TIRER;
       (* elle fait placer dans le tiroir n° Compteur de l'étagère Tirages un nombre au hasard entre
       1 et NNP et qui soit à coup sûr différent des contenus des tiroirs précédents.*)

var Egalite : boolean;
       (* vraie lorsque le numéro tiré est identique à l'un des précédents *)

procedure VERIFIER;
       (* elle fait placer vrai dans la variable Egalite si le contenu du tiroir n° Compteur de
       Tirages est identique à l'un des contenus des tiroirs précédents et faux sinon.*)

begin
repeat
  Tirages[Compteur]:= random(NNP)+1;
  VERIFIER;
until not Egalite;
end;
```

1.6.2 Analyse de VERIFIER

Nous avons à nouveau isolé une tâche (et un problème de programmation) en soi. Nous pouvons oublier tout le reste pour nous concentrer sur ce seul et petit problème. Retour à la case départ, donc.

1.6.2.1 VERIFIER ? Comment faire ?



La stratégie est immédiate

- On va successivement comparer le tiroir n° *Compteur* aux précédents
- Ces comparaisons s'arrêtent dès qu'une égalité est signalée (*Egalite* devenant alors vraie) ou au plus tard avec le tiroir précédent celui portant le n° *Compteur*.
- Lorsqu'aucune égalité n'est détectée, *Egalite* prend la valeur faux.

1.6.2.2 VERIFIER ? Comment faire faire ?

? Quelle marche à suivre proposeriez-vous pour rendre compte de cette stratégie ?

L'essentiel est de traduire la boucle des comparaisons successives qui s'arrêteront à la détection d'une égalité ou lorsque tous les tiroirs précédents celui qui est testé ont été passés en revue.

On peut proposer :

Place 0 dans *C* — *C* de type entier

<u>Répéter</u>

Place succ(*C*) dans *C*

Egalite ← *Tirages* [*C*] = *Tirages* [*Compteur*]

<u>jusqu'à ce que</u> *C*=pred(*Compteur*) ou *Egalite*

1. On constate qu'une nouvelle variable de type "compteur" est indispensable pour écrire la boucle d'exploration. Nous sommes libres du choix de son nom : le seul qui nous soit interdit est *Compteur*, puisque cette variable préexiste et doit comme telle pouvoir être consultée (Cf. le rectangle spécifiant VERIFIER ci-dessus). Nous avons donc baptisé *C* ce compteur.
2. La manière dont la comparaison est écrite peut sembler surprenante :

   Egalite ← *Tirages* [*c*] = *Tirages* [*Compteur*]

   Il n'y a pourtant rien que de très normal : on commande de remplir une variable booléenne (qui ne peut donc accueillir que les valeurs vrai ou faux) et l'information qu'on demande d'y placer (*Tirage* [*C*] = *Tirages* [*Compteur*]) est bien une expression booléenne qui aura forcément les valeurs vrai ou faux. (A rapprocher de la page 39).

?
?
?

L'affectation

Egalite ← *Tirages* [*C*] = *Tirages* [*Compteur*]

est elle équivalente à

<u>Si</u> *Tirages*[*C*] = *Tirages*[*Compteur*] <u>alors</u>

Egalite ← vrai

Disposant de la description de la manière dont l'action VERIFIER sera menée à bien, nous somme en mesure d'écrire

1.6.2.3 VERIFIER ? Comment dire ?

```
procedure VERIFIER;
       (* elle fait placer vrai dans la variable Egalite si le contenu du tiroir n° Compteur de
       Tirages est identique à l'un des contenus des tiroirs précédents et faux sinon.*)

var C : integer;
       (* compteur de boucle *)

begin
C:=0;
Repeat
  C:=succ(C);
  Egalite:=Tirages[C]=Tirages[Compteur];
until (C=pred(Compteur)) or Egalite;
end;
```

1.6.3 Analyse de AFFICHER

Comme toujours, le "Quoi faire ?" a été précisé dans le rectangle de spécification (Cf. page 71).

1.6.3.1 AFFICHER ? Comment faire ?

Il suffira d'annoncer les 6 (en vérité, plutôt pred(NNT)) premiers numéros contenus dans *Tirages*, et le complémentaire comme étant *Tirages* [NNT].

1.6.3.2 AFFICHER ? Comment faire faire ?

Ici aussi une variable du genre compteur sera indispensable. Mais en relisant les spécifications de AFFICHER et particulièrement la liste des variables consultées (page 71), on s'aperçoit que seule *Tirages* est citée.

On a dès lors parfaitement le droit d'appeler *Compteur* la nouvelle variable à définir pour écrire la boucle permettant l'affichage. On verra que cette variable *Compteur* "locale" n'a rien à voir avec la variable *Compteur* définie lors de l'analyse de 1er niveau, dont on ignore en principe totalement l'existence ici.

> Rappelons une fois de plus que les seules contraintes imposées à ceux qui auraient à analyser un module (comme AFFICHER) sont celles reprises dans le rectangle descriptif qui précise exactement quelles sont les variables imposées. Pour les variables définies localement lors de l'analyse, on a donc toute liberté de choix, mis à part l'obligation d'éviter de les nommer comme ces variables imposées.

On écrira donc :

Efface l'écran

Affiche 'Voici les résultats :' (en ligne 2, colonne 4)

Pour *Compteur* allant de 1 à pred(NNT) — *Compteur* de type entier

  Affiche *Tirages* [*Compteur*] (en ligne 6, sans passage à la ligne)

Affiche 'et le numéro complémentaire', *Tirages* [NTT] (en ligne 10, colonne 4)

qui se traduit en :

1.6.3.3 AFFICHER ? Comment dire ?

```
procedure AFFICHER;
(*     elle fait afficher les contenus des tiroirs de Tirages sous la forme d'un écran qui respecte
       les spécifications générales*)
var Compteur : integer;

begin
gotoxy(4,2);
write('Voici les résultats du Lotto :');
gotoxy(8,6);
for Compteur:=1 to pred(NNT) do
  write(Tirages[Compteur]:4);
gotoxy(4,10);
write('et le numéro complémentaire : ', Tirages[NNT]);
end;
```

Nous avons gardé pour la fin le plus facile :

1.6.4 Analyse de AVERTIR

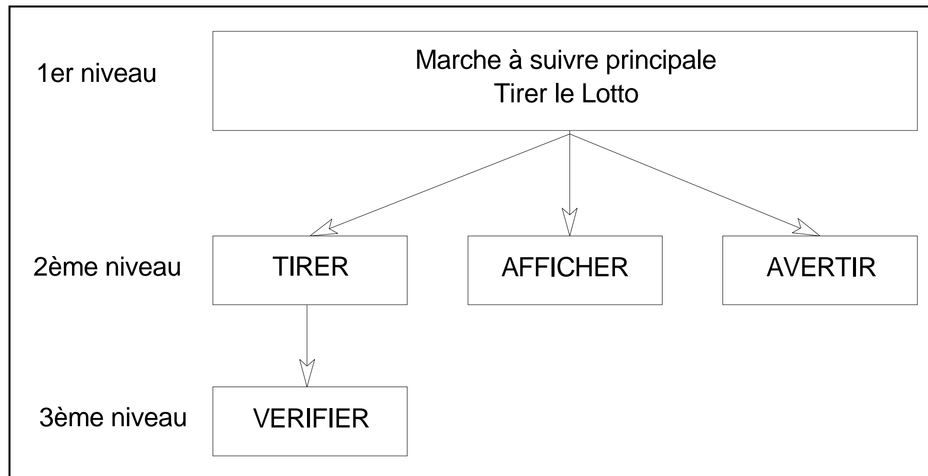
On peut très certainement se contenter de

1.6.4.1 AVERTIR ? Comment dire ?

```
procedure AVERTIR;
       (* Fait afficher l'écran - titre (cf. les spécifications générales) et attend la frappe
       d'Entrée avant d'effacer)*)
begin
clrscr;
writeln('Je vais procéder pour vous au tirage des numéros du Lotto');
write('Frappez Entrée');
readln; (* lecture de la frappe de la touche Entrée *)
clrscr;
end;
```

1.7 L'ensemble de la solution proposée

1. L'architecture générale est à présent visible; elle se marque par une hiérarchie, ou une généalogie entre les divers niveaux d'analyse :



On l'a constaté, cette hiérarchie se traduit en Pascal par l'emboîtement (à différents niveaux) des procédures.

Ainsi le programme principal donne naissance à trois procédures-filles (TIRER, AFFICHER et AVERTIR), TIRER ayant elle-même une procédure-fille VERIFIER.

La filiation se marque en Pascal par le fait que la procédure fille est enclose (ou imbriquée) dans la procédure-mère (ou le programme principal), entre la partie déclaration (const, type, var) et le begin de la partie exécutable.

L'ordre dans lequel les textes des procédures soeurs se présentent est pour l'instant sans importance. Il ne faut en tout cas pas le confondre avec l'ordre dans lequel ces procédures sont appelées.

Ici, la structure d'imbrication est donc la suivante (et ceci me donne l'occasion de présenter le texte du programme dans son entièreté) :

```
program LOTTO;
        (* Il fait simuler un tirage du lotto. On y utilise un tableau
        à 7 composantes pour y stocker les résultats obtenus *)
uses WinCRT;
        (* pour pouvoir utiliser les procédures de gestion d'écran
        comme Clrscr ou Gotoxy en Turbo Pascal sous Windows *)
const NNP = 42; (* plus grand numéro pouvant être obtenu *)
       NNT = 7;  (* nombre de numéros tirés (y compris le
        complémentaire) *)

type NumerosPossibles = 1..NNP;

var Compteur: integer;
        (* compteur pour les boucles for *)
     Tirages : array[1..NNT] of NumerosPossibles; (* qui
        contiendra les numéros obtenus *)
     Reponse : char;
        (* réponse de l'utilisateur *)
```

```
procedure TIRER;
       (* elle fait placer dans le tiroir n° Compteur de
       l'étagère Tirages un nombre au hasard entre 1 et
       NNP et qui soit à coup sûr différent des contenus
       des tiroirs précédents.*)

var Egalite : boolean;
       (* vraie lorsque le numéro tiré est
       identique à l'un des précédents *)
     procedure VERIFIER;
           (* elle fait Placer vrai dans la
           variable Egalite si le contenu du
           tiroir n° Compteur de Tirages est
           identique à l'un des contenus des
           tiroirs précédents et faux sinon.*)

     var C : integer;
     (* compteur de boucle *)

     begin
     C:=0;
     Repeat
       C:=succ(C);
       Egalite:=
           Tirages[C]=Tirages[Compteur];
     until (C=pred(Compteur)) or Egalite;
     end;

begin
repeat
  Tirages[Compteur]:= random(NNP)+1;
  VERIFIER;
until not Egalite;
end;
```

```
procedure AFFICHER;
       (* elle fait afficher les contenus des tiroirs de
       Tirages sous la forme d'un écran qui respecte les
       spécifications générales*)
var Compteur : integer;

begin
gotoxy(4,2);
write('Voici les résultats du Lotto :');
gotoxy(8,6);
for Compteur:=1 to pred(NNT) do
  write(Tirages[Compteur]:4);
gotoxy(4,10);
write('et le numéro complémentaire : ',
       Tirages[NNT]);
end;
```

```
procedure AVERTIR;
       (* Fait afficher l'écran - titre (cf. les
       spécifications générales) et attend la frappe
       d'Entrée avant d'effacer)*)
begin
clrscr;
writeln('Je vais procéder pour vous au tirage des
       numéros du Lotto');
write('Frappez Entrée');
readln; (* lecture de la frappe de la touche Entrée *)
clrscr;
end;
```

```
begin
randomize;
AVERTIR;
repeat
  ClrScr; (* pour effacer avant de recommencer un tirage *)
  Tirages[1]:=random(NNP)+1;
  for Compteur:=2 to NNT do
     TIRER;
  AFFICHER;
  gotoxy(4,20);
  write(' Recommence-t-on un tirage (O ou N)?');
  readln(Reponse);
  Reponse:= upcase(Reponse);
until Reponse = 'N';
end.
```

2. On l'a constaté, il est possible de définir au sein d'une procédure des variables spécifiques (on dit souvent locales) (comme *Egalite* dans TIRER ou *C* dans VERIFIER). On le verra dans un instant, ces variables n'existeront que pendant le temps d'exécution des procédures qui les définissent (et des procédures-filles activées lors de cette exécution). Il serait évidemment possible de la même manière de définir des constantes ou des types spécifiques à une procédure, au sein de la partie déclaration de cette procédure.

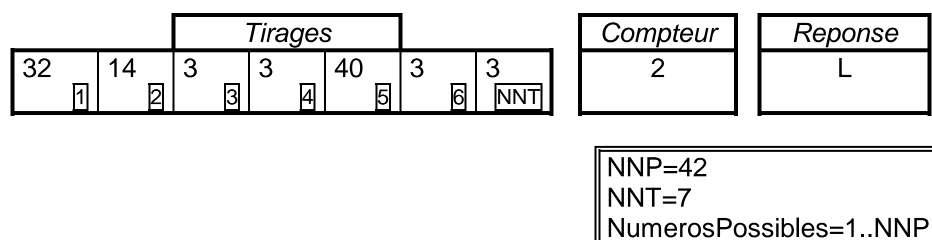
1.8 Comment se passe l'exécution de l'ensemble ?

Nous allons, une fois de plus, en donner une description métaphorique et imagée, qui va faire intervenir non plus un, mais plusieurs exécutants, chacun étant accompagné de son "installateur d'étiquettes".

Disons, dès à présent, que chaque procédure aura son exécutant propre (disposant de la partie exécutable du texte de la procédure concernée) et son "installateur d'étiquettes" (disposant de la partie déclaration de cette même procédure).

1. L'exécution commence par le travail du couple exécutant-installateur chargé du programme principal. Ce dernier peut être trouvé sous forme "marche à suivre" à la page 69 et sous forme "programme" à la page 73. Il s'agit bien du seul texte (déclaration et corps du programme) du programme principal et non du texte global comportant également les textes des procédures.

   L'installateur d'étiquettes note dans un coin du local les constantes (NTT = 7, NNP = 42) et les types (ici, un seul type, à savoir NumerosPossibles = 1..NNP), **afin que ces définitions soient disponibles tout au long de l'exécution**. Il réserve ensuite les casiers (*Compteur* et *Reponse*) et l'étagère (*Tirages*) indiqués (dans la partie déclaration dont il dispose) et y colle les étiquettes mentionnées. Le local ressemble alors à



   On remarquera que, comme d'habitude, des informations (du type approprié) traînent dans les casiers et les tiroirs.

2. Cela fait, c'est l'exécutant-principal qui commence son travail. La première instruction qu'il rencontre, (outre randomize si on se réfère au texte du programme et qui est sans grand intérêt ici) est AVERTIR; elle ne fait pas partie des instructions élémentaires qu'il "comprend", il se tourne donc vers un coin du local où des exécutants-auxiliaires sont en train d'attendre, chacun accompagné de son installateur d'étiquettes. Il a devant les yeux trois candidats : TIRER, AVERTIR, et AFFICHER.

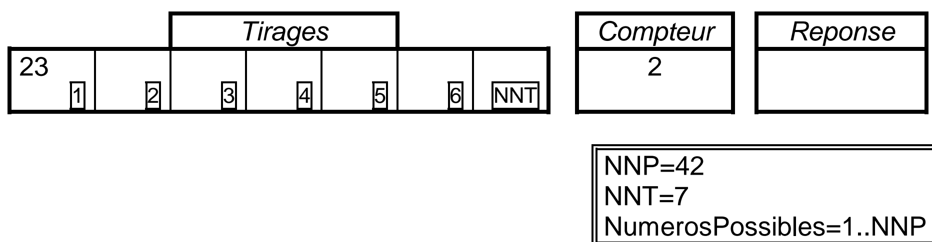
   Il appelle donc le couple chargé d'AVERTIR et se retire dans le coin opposé du local. Le couple chargé d'AVERTIR vient alors faire le travail mentionné par le texte de la procédure correspondante. A l'issue de ce travail (que je ne détaille pas ici car sa simplicité ne nous permettrait pas d'illustrer les faits importants), il "rend la main" à celui qui avait fait appel à ses services, l'exécutant-principal.

3. Ce dernier poursuit l'exécution là où il s'était interrompu lorsqu'il avait appelé AVERTIR.

   Il va donc, comme commandé par le programme principal :

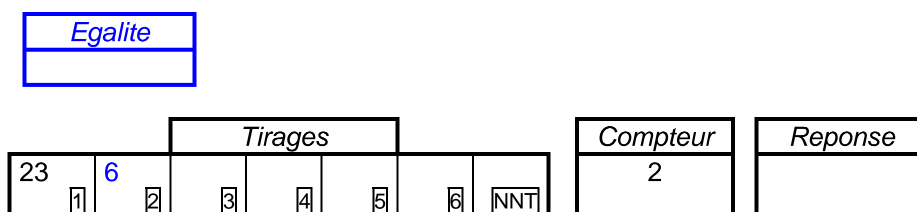
   - placer un nombre au hasard dans le premier tiroir de *Tirages*;
   - donner (comme le demande la boucle POUR...) la valeur 2 à *Compteur*.

   Il tombe alors sur l'ordre TIRER. A nouveau, il se tourne vers le coin du local où les exécutants sont en attente et appelle le couple chargé de TIRER en lui laissant un local :



   où cette fois, pour la facilité, les casiers et tiroirs dont le contenu est non significatif ont été dessiné vides.

4. L'installateur (qui dispose de la partie déclaration) de TIRER (Cf. page 76), installe comme demandé une étiquette *Egalite* sur un casier de type booléen. C'est alors l'exécutant de TIRER qui commence le travail exigé par la partie exécutable de TIRER. Comme indiqué, il consulte *Compteur* (contenant 2) et place un nombre au hasard (ici 6) dans le tiroir numéro *Compteur* (2) de *Tirages*.[10] L'instruction suivante est VERIFIER. L'exécutant de TIRER se tourne à son tour vers le coin du local où se tient le couple chargé de VERIFIER. Il appelle donc ce couple et rejoint dans le coin opposé, l'exécutant-principal toujours en train d'attendre, laissant aux arrivants un local :

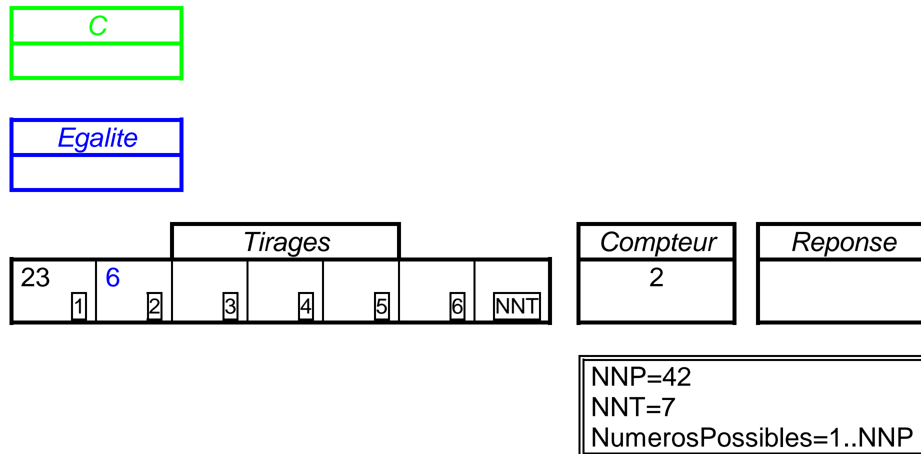


10 Si vous disposez du docuemnt "en couleur", le travail de TIRER est présenté en bleu et ce lui de VERIFIER en vert sur les schémas qui suivent.

 © CeFIS - FUNDP 2002

NNP=42
NNT=7
NumerosPossibles=1..NNP

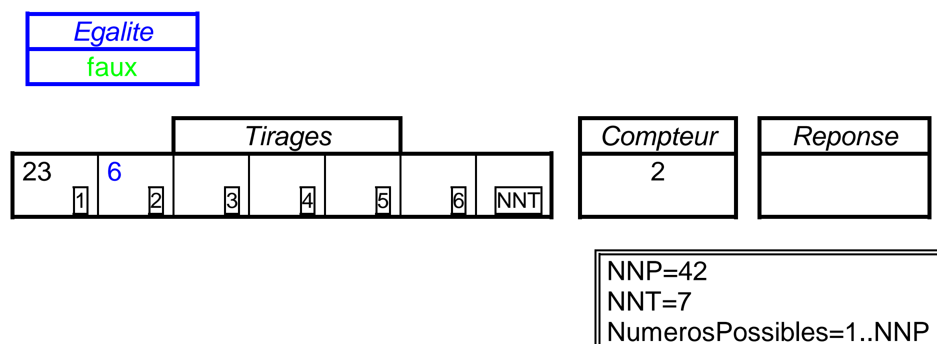
5. L'installateur de VERIFIER colle, comme cela est exigé dans la partie déclaration de la procédure VERIFIER (page 77), l'étiquette *C* sur un casier entier et le travail de l'exécutant VERIFIER commence alors avec un local :



L'exécutant, comme indiqué dans la partie exécutable de VERIFIER (page 77) :

- place 0 dans *C*;
- remplace ce contenu de *C* par la valeur suivante (succ(*C*)). *C* contient alors 1;
- évalue la valeur de l'expression booléenne *Tirages* [*C*] = *Tirages* [*Compteur*] (en l'occurence ici la valeur de *Tirages* [1] = *Tirages* [2], soit faux) et place le résultat de l'évaluation dans *Egalite*;
- il évalue alors la condition *C*=pred(*Compteur*) ou *Egalite*; comme elle est vraie (puisque *C*=pred(*Compteur*) la boucle Répéter s'arrête là.

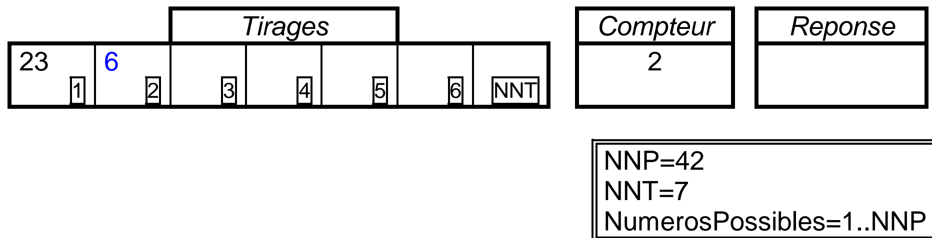
Le travail de l'exécutant chargé de VERIFIER s'arrête donc là. Son installateur d'étiquettes **reprend l'étiquette *C*** qu'il avait fixée et tous deux retournent dans le coin du local d'où ils venaient, non sans avoir préalablement rendu la main à l'exécutant de TIRER qui retrouve donc un local qui a l'aspect suivant :



où ne subsiste plus aucune trace de l'étiquette *C* installée par le couple chargé de VERIFIER.

6. L'exécutant de TIRER poursuit l'exécution de sa marche à suivre (page 75 ou 76) interrompue par l'appel de VERIFIER; il lui reste seulement à tester la valeur de la condition not *Egalite*. La condition étant vraie, son travail s'interrompt. Il prévient son installateur d'étiquettes qui **reprend l'étiquette *Egalite*** qu'il avait apposée et tous deux retournent attendre dans le coin du local en rendant le contrôle à l'exécutant-principal.

7. Ce dernier retrouve un local qui lui est familier, le seul changement visible étant le remplissage du tiroir n° 2 de *Tirages* :



Il lui reste (page 69 ou 73) à incrémenter la valeur du *Compteur* (qui vaut alors 3) et à constater que cette valeur ne dépasse pas NNT (test inhérent à la boucle Pour). Comme précédemment, il appelle alors TIRER pour la suite du travail.

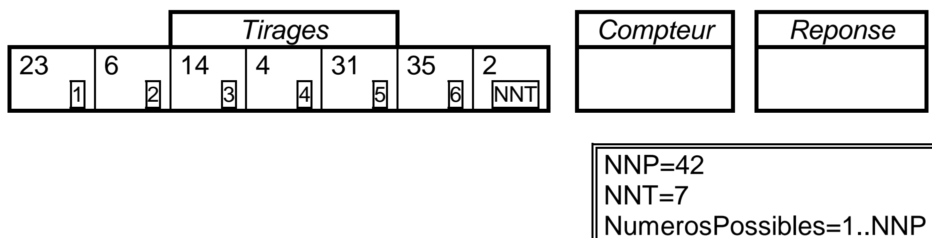
8. Le ballet se poursuit avec une nouvelle arrivée du couple chargé de TIRER et l'installation de l'étiquette *Egalite* sur un casier de type booléen (pas nécessairement le même que précédemment).

   Ce sera ensuite l'appel de VERIFIER, l'installation de l'étiquette *C*, le travail de l'exécutant puis le départ en emportant l'étiquette *C*.

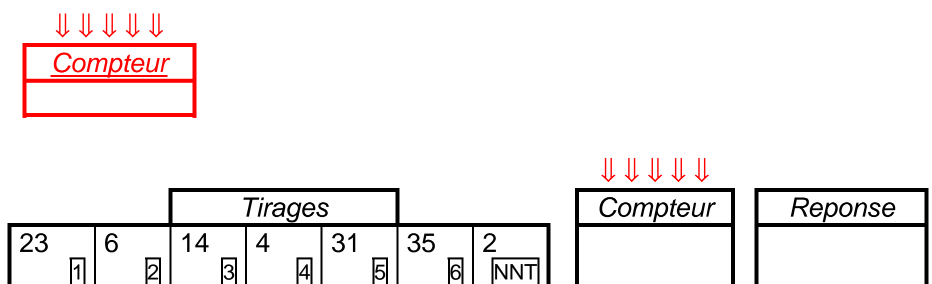
   Le travail de TIRER se poursuivra avec soit le rappel de VERIFIER après un nouveau tirage (si le précédent était inadéquat), soit la fin du travail (de TIRER) et le retour de l'exécutant-principal.

9. Le cycle se poursuivra jusqu'à ce que *Tirages* soit correctement garnie. Il reste un détail important à signaler : lorsque l'exécutant-principal, face à la commande AFFICHER, appelle le couple qui en est chargé,

   ce dernier découvre un local :



Comme exigé (page 78), l'installateur colle sur un casier de type entier une nouvelle étiquette *Compteur* et l'exécutant chargé de AFFICHER se met au travail. Il est fait mention du casier *Compteur* dans le texte de la marche à suivre dont il dispose. Et face à lui, deux casiers étiquetés *Compteur* se présentent : le sien (repéré par une étiquette rouge et soulignée) et celui précédemment installé par l'exécutant-principal



NNP=42 NNT=7 NumerosPossibles=1..NNP

Le choix sera alors toujours identique : c'est du plus récemment installé (celui qui est "le plus haut" avec le type de schéma proposé où les casiers les plus récemment étiquetés sont dessinés au dessus) dont il est forcément question. C'est donc avec **son** casier *Compteur* que le travail sera effectué. Le seul casier alors tout à fait inaccessible à l'exécutant chargé d'AFFICHER est le *Compteur* défini par l'installateur-principal (et situé plus bas).

Cette description imagée touche donc à des concepts importants de la programmation : variables globales et locales, portée d'une variable, appel de procédure...

Notons encore que le comportement décrit est celui prescrit par le langage Pascal, mais que d'autres langages seraient redevables des mêmes explications.

Ainsi donc,

- Chaque procédure peut comporter une partie déclaration de variables (et de constantes et de types). Ces déclarations restent valables tant que l'exécutant-auxiliaire chargé de cette procédure ne clôture pas son travail : soit qu'il soit lui-même au travail, soit qu'il ait fait appel à son tour à d'autres auxiliaires.
- A la fin de l'exécution de la procédure, toutes les déclarations locales effectuées disparaissent. Dans la métaphore employée, les étiquettes (mais ce serait aussi le cas des déclarations de constantes et de type) cessent d'exister lorsque l'exécutant retourne dans le coin du local d'où il venait, rendant alors la main à l'exécutant qui l'avait appelé.
- Lorsque plusieurs variables portant le même nom sont présentes pendant le déroulement d'une procédure, c'est toujours la plus récemment installée qui prévaut.

Ces trois règles rendent compte de la manière dont un programme Pascal est exécuté au fil des appels successifs. Classiquement, on emploie les termes de variables globales pour celles définies au sein du programme principal et de variables locales pour celles définies au sein des diverses procédures. Il faut regretter cette terminologie : plutôt qu'une connotation spatiale comme celle apportée par les mots "local" ou "global", c'est d'une connotation temporelle dont nous avons besoin; plutôt que "globale" et "locale", c'est de "permanente" et "temporaire" qu'il faudrait parler.

Le terme local (ou temporaire) doit d'ailleurs être pris dans un sens relatif et non absolu. Ce qui est "local" pour une procédure est bien entendu "global" pour les procédures (filles et autres descendantes) qu'elle appelle. On parle également de portée d'une variable en évoquant la portion du programme où elle peut être invoquée et utilisée. La description imagée fournie ci-dessus suffit pour décider aisément où (ou plutôt quand) l'emploi d'une variable est permis : pendant l'exécution des procédures filles appelées (et des autres descendants appelés ensuite).

Ainsi donc, une déclaration faite au sein du programme principal ou d'une procédure est valable pour toutes les procédures "descendantes" (filles, petites-filles, ...). Autrement dit, lorsque l'exécutant d'une procédure commence son travail, il a accès à toutes les variables de ses ascendantes directes (et bien entendu du programme principal), hormis celles dont les noms sont identiques, auquel cas, seule la plus récente est accessible. Et il en va de même de toutes les déclarations.

Nous verrons dans la suite "qui peut appeler qui au sein" de la généalogie des procédures constituant l'ensemble d'un programme.

Retenons seulement pour l'instant que le programme principal ou une procédure peut seulement appeler ses procédures filles, à l'exclusion des autres descendantes. Pas question donc, dans le problème du Lotto traité ci-dessus que le programme principal puisse appeler la procédure VERIFIER (qui est une petite fille et non une fille) (Cf. page 79).

1.9 Comment faire faire à un premier niveau (pour la seconde stratégie) ?

Forts de la première analyse menée à son terme (page 65), nous sommes en mesure de préciser la

1.9.1 Structure des données

1.9.1.1 Constantes

Comme précédemment, 2 constantes sont indispensables : forcément, ces constantes sont attachées à la description même de ce qu'est le Lotto et non à l'analyse déjà menée. Je garde donc, avec la même signification que ci-dessus, NNP et NNT.

Je définirai à nouveau le type NumerosPossibles comme l'intervalle des entiers entre 1 et NNP.

1.9.1.2 Tableau nécessaire

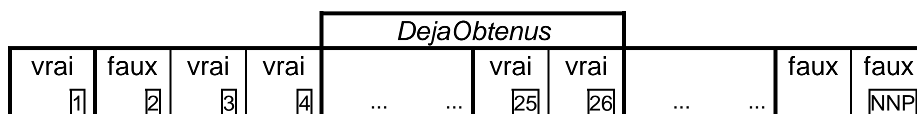
? Quel est ici, pour rendre compte de la seconde manière de procéder, le tableau à définir ? Avec combien de tiroirs et quelles étiquettes ?

Pour calquer le "Comment faire faire ?" sur la seconde stratégie développée, il nous faut à présent une étagère à NNP tiroirs. Nous marquerons un numéro (non encore marqué) en changeant le contenu du tiroir portant ce numéro.

L'idéal serait donc de choisir une étagère booléenne (chaque tiroir contenant seulement soit vrai soit faux). Par ailleurs, ceci nous permettra d'approfondir les traitements mettant en oeuvre les booléens.

> On peut bien entendu choisir ici un tableau de type entier, caractère, ...Ceci conduit à une structure de solution identique mais à quelques changements d'écritures.

Ainsi donc, une étagère étant nécessaire, je la baptiserai *DejaObtenus* : quand le tiroir d'étiquette N sera (= contiendra) vrai, c'est que le nombre N aura été "déjà obtenu"; si tiroir est (contient) faux, N n'est pas "déjà obtenu".



L'étagère *DejaObtenus* comporte comme étiquettes la suite des NumerosPossibles (entre 1 et NNP). Les NNP tiroirs sont tous de type booléen.

En Pascal : `DejaObtenus : array[NumerosPossibles] of boolean`.

1.9.2 Marche à suivre

Je vais évidemment la calquer autant que possible sur celle de la première stratégie. Il faut cependant être attentif à trois détails :

- l'étagère *DejaObtenus* doit avoir tous ses tiroirs faux au début;

 © CeFIS - FUNDP 2002

- la vérification du fait qu'un tirage a déjà été obtenu est bien plus facile;
- le complémentaire est traité à part, tant lors des tirages que lors de l'affichage.

Tout ceci conduit à proposer :

FAIS TOUT CE QU'IL FAUT POUR PRESENTER A L'UTILISATEUR UN ECRAN D'AVERTISSEMENT (CF LES SPÉCIFICATIONS) QUI DEVRA S'EFFACER A LA FRAPPE DE ENTREE.

Répéter

FAIS TOUT CE QU'IL FAUT POUR INITIALISER L'ETAGERE *DejaObtenus* EN PLAÇANT faux DANS TOUS LES TIROIRS.

Place vrai dans le tiroir de l'étagère *DejaObtenus* portant comme numéro un nombre au hasard entre 1 et NNP.

Pour *Compteur* allant de 2 à pred(NNT) — *Compteur* de type entier

FAIS TOUT CE QU'IL FAUT POUR MARQUER vrai UN TIROIR AU HASARD DE *DejaObtenus* ET QUI NE CONTENAIT PAS DEJA VRAI.

FAIS TOUT CE QU'IL FAUT POUR PLACER DANS *Complementaire* UN NOMBRE AU HASARD ENTRE 1 ET NNP NON ENCORE OBTENU — *Complementaire* de type NumerosPossibles

FAIS TOUT CE QU'IL FAUT POUR AFFICHER LES RESULTATS TELS QUE MARQUES DANS *DejaObtenus* ET POUR AFFICHER LE *Complementaire*.

Demande à l'utilisateur s'il veut recommencer et place sa réponse dans *Reponse*. — *Reponse* de type caractère

jusqu'à ce que *Reponse* = 'N'

Comme précédemment un *Compteur* est nécessaire (pour le déroulement de la boucle de remplissage des tiroirs de 2 à pred(NNT); nous le choisissons bien entendu de type entier (et même compris entre 2 et NNT). Une variable *Reponse* est évidemment indispensable aussi.

Il nous faut aussi un casier *Complementaire* qui contiendra le numéro complémentaire. Il est de type NumerosPossibles.

Sous forme abrégée, la marche à suivre peut se réécrire :

AVERTIR

Répéter

INITIALISER

Place vrai dans le tiroir de l'étagère *DejaObtenus* portant comme numéro un nombre au hasard entre 1 et NNP.

Pour *Compteur* allant de 2 à pred(NNT) — *Compteur* de type entier

TIRER_NUMERO

TIRER_COMPLEMENTAIRE — *Complementaire* de type NumerosPossibles

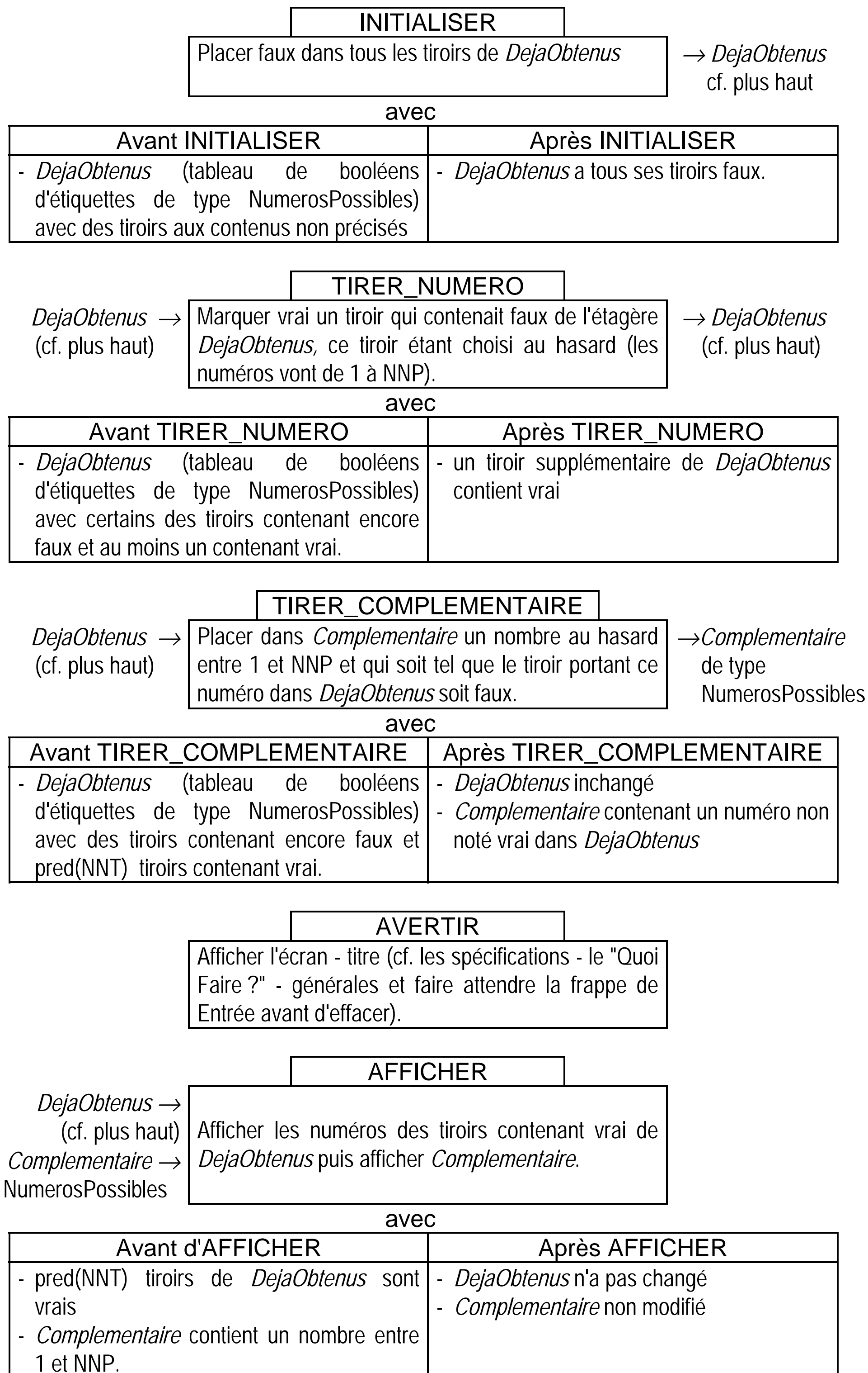
AFFICHER

Demande à l'utilisateur s'il veut recommencer et place sa réponse dans *Reponse*.

jusqu'à ce que *Reponse* = 'N' — *Reponse* de type caractère

Comme lors du développement de la première stratégie, cette analyse de premier niveau n'est complète qu'à la condition que nous précisions exactement ce qu'est chacune des actions complexes non encore décortiquées.

Ainsi :



INITIALISER

Placer faux dans tous les tiroirs de *DejaObtenus* → *DejaObtenus* cf. plus haut

avec

Avant INITIALISER	Après INITIALISER
- *DejaObtenus* (tableau de booléens d'étiquettes de type NumerosPossibles) avec des tiroirs aux contenus non précisés	- *DejaObtenus* a tous ses tiroirs faux.

TIRER_NUMERO

DejaObtenus → (cf. plus haut) Marquer vrai un tiroir qui contenait faux de l'étagère *DejaObtenus*, ce tiroir étant choisi au hasard (les numéros vont de 1 à NNP). → *DejaObtenus* (cf. plus haut)

avec

Avant TIRER_NUMERO	Après TIRER_NUMERO
- *DejaObtenus* (tableau de booléens d'étiquettes de type NumerosPossibles) avec certains des tiroirs contenant encore faux et au moins un contenant vrai.	- un tiroir supplémentaire de *DejaObtenus* contient vrai

TIRER_COMPLEMENTAIRE

DejaObtenus → (cf. plus haut) Placer dans *Complementaire* un nombre au hasard entre 1 et NNP et qui soit tel que le tiroir portant ce numéro dans *DejaObtenus* soit faux. →*Complementaire* de type NumerosPossibles

avec

Avant TIRER_COMPLEMENTAIRE	Après TIRER_COMPLEMENTAIRE
- *DejaObtenus* (tableau de booléens d'étiquettes de type NumerosPossibles) avec des tiroirs contenant encore faux et pred(NNT) tiroirs contenant vrai.	- *DejaObtenus* inchangé - *Complementaire* contenant un numéro non noté vrai dans *DejaObtenus*

AVERTIR

Afficher l'écran - titre (cf. les spécifications - le "Quoi Faire ?" - générales et faire attendre la frappe de Entrée avant d'effacer).

AFFICHER

DejaObtenus → (cf. plus haut) *Complementaire* → : NumerosPossibles Afficher les numéros des tiroirs contenant vrai de *DejaObtenus* puis afficher *Complementaire*.

avec

Avant d'AFFICHER	Après AFFICHER
- pred(NNT) tiroirs de *DejaObtenus* sont vrais - *Complementaire* contient un nombre entre 1 et NNP.	- *DejaObtenus* n'a pas changé - *Complementaire* non modifié

Plus clairement encore que précédemment, cette écriture des spécifications précises de chacune des mini-actions "complexes" laissées non analysées dans la marche à suivre principale est fastidieuse, tant le "Comment faire faire ?" correspondant est immédiat.

Même dans ces cas simples cependant, il faut se forcer à l'écriture des spécifications. Programmer (au sens large de ce mot), c'est avant tout spécifier. (Cf. le chapitre 1 du volume 1 de "Images pour programmer".).

? Que pensez-vous de la marche à suivre suivante, les actions complexes étant décrites exactement comme ci-dessus ?

AVERTIR
Répéter
    INITIALISER
    Pour *Compteur* allant de **1** à pred(NNT)
        TIRER_NUMERO
    TIRER_COMPLEMENTAIRE
    AFFICHER
    Demande à l'utilisateur s'il veut recommencer et place sa réponse dans *Reponse*.
jusqu'à ce que *Reponse* = 'N'

1.10 Comment dire ? (l'analyse de 1er niveau de la 2ème stratégie)

```
program LOTTO;
(* il fait simuler un tirage du lotto *)
(* on y utilise  un tableau de booléens pour signaler quels numéros sont tirés *)
uses WinCRT;
const NNP = 42;
       (* Nombre de numéros possibles *)
      NNT = 7;
       (* Nombre de numéros tirés (y compris le complémentaire) *)

type   NumerosPossibles = 1..NNP;

var    Compteur: integer;
       (* compteur pour la boucle for *)
       DejaObtenus : array[NumerosPossibles] of boolean;
       (* une composante donnée sera vraie (true) si le numéro correspondant a déjà été tiré et
       fausse (false) sinon *)
       Complementaire : NumerosPossibles;
       (* numéro complémentaire *)
       Reponse : char;
       (* réponse de l'utilisateur à la question de savoir si on continue*)

procedure AVERTIR;
       (* Fait afficher l'écran - titre (cf. les spécifications générales) et attend la frappe
       d'Entrée avant d'effacer)*)

procedure INITIALISER;
       (* elle fait placer faux dans tous les tiroirs de DejaObtenus *)

procedure AFFICHER;
       (* elle fait Afficher les numéros des tiroirs vrais de DejaObtenus puis afficher Comple-
       mentaire.*)

procedure TIRER_NUMERO;
       (* elle fait marquer vrai un tiroir qui contenait faux de l'étagère DejaObtenus ce tiroir
       étant choisi au hasard (les numéros vont de 1 à NNP).*)

procedure TIRER_COMPLEMENTAIRE;
       (* elle fait Placer dans Complementaire un nombre au hasard entre 1 et NNP et qui soit tel
       que le tiroir portant ce numéro dans DejaObtenus soit faux.*)

begin (* début du programme principal *)
randomize;
AVERTIR;
repeat
  INITIALISER;
  DejaObtenus[random(NNP)+1]:=true;
  for Compteur:=2 to pred(NNT) do
      TIRER_NUMERO;
  TIRER_COMPLEMENTAIRE;
```

```
  AFFICHER;
  gotoxy(4,20);
  write(' Recommence-t-on un tirage (O ou N)?');
  readln(Reponse);
  Reponse:= upcase(Reponse);
until Reponse = 'N';
end.
```

1.11 Analyse de second niveau

1.11.1 Analyse de INITIALISER

Le "comment faire ?" est ici immédiat et nous pouvons rapidement écrire

1.11.1.1 INITIALISER ? Comment faire faire ?

Pour *Compteur* allant de 1 à NNP — *Compteur* de type NumerosPossibles
  Place faux dans le tiroir n° *Compteur* de l'étagère *DejaObtenus*

C'est, comme d'habitude lorsqu'il s'agit de parcourir un tableau, une boucle Pour ... qui convient, le compteur de boucle pouvant à nouveau s'appeler *Compteur* (et étant un entier de type NumerosPossibles).

1.11.1.2 INITIALISER ? Comment dire ?

```
procedure INITIALISER;
      (* elle fait placer faux dans tous les tiroirs de DejaObtenus *)
var   Compteur : NumerosPossibles;
      (* compteur de boucle *)
begin
for Compteur := 1 to NNP do
    DejaObtenus[Compteur]:=false;
end;
```

1.11.2 Analyse de TIRER_NUMERO

1.11.2.1 TIRER_NUMERO ? Comment faire ?

Comme l'analyse menée page 67 le montrait, il nous faut effectuer un tirage et vérifier si le tiroir de *DejaObtenus* portant ce numéro ne contiendrait pas déjà vrai, auquel cas le tirage devrait être recommencé. Une fois un numéro acceptable obtenu, on le retient en faisant passer à vrai le contenu du tiroir de *DejaObtenus* portant ce numéro.

1.11.2.2 TIRER_NUMERO ? Comment faire faire ?

Répéter
  Place un nombre au hasard entre 1 et NNP dans *Resultat*
jusqu'à ce que le tiroir n° *Resultat* de *DejaObtenus* soit faux
Place vrai dans le tiroir n° *Resultat* de *DejaObtenus*

Resultat de type NumerosPossibles

Une variable *Resultat* est indispensable pour contenir le nombre tiré. Elle est de type NumerosPossibles. Sous forme abrégée :

Répéter
  Resultat ← un nombre au hasard entre 1 et NNP
jusqu'à ce que non *DejaObtenus*[*Resultat*] — *Resultat* de type NumerosPossibles
DejaObtenus[*Resultat*] ← vrai

1.11.2.3 TIRER_NUMERO ? Comment dire ?

```
procedure TIRER_NUMERO;
```

```
       (* elle fait marquer vrai un tiroir qui contenait faux de l'étagère DejaObtenus ce tiroir
       étant choisi au hasard (les numéros vont de 1 à NNP).*)
var    Resultat : NumerosPossibles;
       (* pour accueillir les tirages éventuellement recommencés *)
begin
repeat
   Resultat:= random(NNP)+1;
until not DejaObtenus[Resultat];
DejaObtenus[Resultat]:=true;
end;
```

1.11.3 Analyse de TIRER_COMPLEMENTAIRE

1.11.3.1 TIRER_COMPLEMENTAIRE ? Comment faire ?

La stratégie de tirage et de validation est exactement celle ci-dessus. Le seul changement est la manière de retenir le complémentaire : on placera le nombre retenu dans *Complementaire* plutôt que de marquer le tiroir correspondant de *DejaObtenus*.

? Quelle marche à suivre proposeriez vous ?

1.11.3.2 TIRER_COMPLEMENTAIRE ? Comment faire faire ?

Répéter
    Place un nombre au hasard entre 1 et NNP dans *Complementaire*
jusqu'à ce que le tiroir n° *Complementaire* de *DejaObtenus* soit faux

La variable *Complementaire* peut évidemment être utilisée pour les éventuels tirages répétés.

1.11.3.3 TIRER_COMPLEMENTAIRE ? Comment dire ?

```
procedure TIRER_COMPLEMENTAIRE;
       (* elle fait Placer dans Complementaire un nombre au hasard entre 1 et NNP et qui soit tel
       que le tiroir portant ce numéro dans DejaObtenus soit faux.*)
begin
repeat
    Complementaire := random(NNP)+1;
until not DejaObtenus[Complementaire];
end;
```

1.11.4 Analyse de AFFICHER

On peut pratiquement reprendre l'analyse menée pour la première stratégie (page 78) sauf l'affichage du complémentaire, qui s'effectue simplement sur base du contenu de *Complementaire*.

On peut donc proposer immédiatement :

1.11.4.1 AFFICHER ? Comment dire ?

```
procedure AFFICHER;
       (* elle fait Afficher les numéros des tiroirs vrais de DejaObtenus puis afficher Comple-
       mentaire.*)
var Compteur : integer;

begin
clrscr;
gotoxy(4,2);
write('Voici les résultats du Lotto :');
gotoxy(8,6);
for Compteur:=1 to NNP do
  if DejaObtenus[Compteur] then
    write(Compteur:4);
gotoxy(4,10);
write('et le numéro complémentaire : ', Complementaire);
end;
```

Quant à la procédure AVERTIR, rien ne change.

Ceci conduit à une structure générale du programme :

```
program LOTTO;
(* il fait simuler un tirage du lotto *)
(* on y utilise  un tableau de booléens pour signaler quels numéros
       sont tirés *)
uses WinCRT;
const NNP = 42;
       (* Nombre de numéros possibles *)
      NNT = 7;
       (* Nombre de numéros tirés (y compris le complémentaire) *)
type   NumerosPossibles = 1..NNP;

var    Compteur: integer;
       (* compteur pour la boucle for *)
       DejaObtenus : array[NumerosPossibles] of boolean;
       (* une composante donnée sera vraie (true) si le numéro
       correspondant a déjà été tiré et fausse (false) sinon *)
       Complementaire : NumerosPossibles;
       (*numéro complémentaire*)
       Reponse : char;
       (* réponse de l'utilisateur à la question de savoir si on
       continue*)
   procedure AVERTIR;
          (* Fait afficher l'écran - titre (cf. les spécifications
          générales) et attend la frappe d'Entrée avant
          d'effacer)*)
   begin
   clrscr;
   writeln('Je vais procéder pour vous au tirage des
          numéros du Lotto');
   write('Frappez Entrée');
   readln; (* lecture de la frappe de la touche Entrée *)
   clrscr;
   end;

   procedure INITIALISER;
          (* elle fait placer faux dans tous les tiroirs de
          DejaObtenus *)
   var    Compteur : NumerosPossibles;
          (* compteur de boucle *)
   begin
   for Compteur := 1 to NNP do
       DejaObtenus[Compteur]:=false;
   end;

   procedure AFFICHER;
          (* elle fait Afficher les numéros des tiroirs vrais de
          DejaObtenus puis afficher Complementaire.*)
   var Compteur : integer;

   begin
   clrscr;
   gotoxy(4,2);
   write('Voici les résultats du Lotto :');
   gotoxy(8,6);
   for Compteur:=1 to NNP do
     if DejaObtenus[Compteur] then
       write(Compteur:4);
   gotoxy(4,10);
   write('et le numéro complémentaire : ', Complementaire);
   end;
```

```
procedure TIRER_NUMERO;
       (* elle fait marquer vrai un tiroir qui contenait faux
       de l'étagère DejaObtenus ce tiroir étant choisi au
       hasard (les numéros vont de 1 à NNP).*)
var    Resultat : NumerosPossibles;
       (* pour accueillir les tirages éventuellement
       recommencés *)
begin
repeat
   Resultat:= random(NNP)+1;
until not DejaObtenus[Resultat];
DejaObtenus[Resultat]:=true;
end;

procedure TIRER_COMPLEMENTAIRE;
(* elle fait placer dans Complementaire un nombre au hasard
       entre 1 et NNP et qui soit tel que le tiroir portant ce
       numéro dans DejaObtenus soit faux. *)
begin
repeat
    Complementaire := random(NNP)+1;
until not DejaObtenus[Complementaire];
end;

begin (* début du programme principal *)
randomize;
AVERTIR;
repeat
  INITIALISER;
  DejaObtenus[random(NNP)+1]:=true;
  for Compteur:=2 to pred(NNT) do
      TIRER_NUMERO;
  TIRER_COMPLEMENTAIRE;
  AFFICHER;
  gotoxy(4,20);
  write(' Recommence-t-on un tirage (O ou N)?');
  readln(Reponse);
  Reponse:= upcase(Reponse);
until Reponse = 'N';
end.
```

2. Simulation du jeu de poker

2.1 Jeu de poker : description floue

Je souhaite que l'ordinateur, après avoir simulé 1000 lancers successifs de 5 dés, me donne le nombre de paires, de double paires, ... avec le sens qu'ont ces résultats au poker.

2.2 Jeu de poker : Quoi faire ?

2.2.1 Les entrées

Il n'y en aura ici absolument aucune, l'utilisateur n'intervenant que par l'appui sur la touche Entrée à chaque fois que le programme s'interrompra pour permettre de prendre connaissance d'un écran de résultats.

2.2.2 Les traitements

On souhaite 1000 lancers successifs de 5 dés (les faces étant marquées de 1 à 6). Chacun des lancers (de 5 dés) sera affiché (comme précisé ci-dessous). On fournira ensuite les résultats globaux des 1000 lancers selon le nomenclature en vigueur au poker :

- nombre de paires (2 faces identiques lors d'un lancer)

- nombre de double-paires (2 fois 2 faces identiques)
- nombre de brelans (3 faces identiques)
- nombre de fulls (3 faces identiques + 2 faces identiques) (= brelan + paire)
- nombre de carrés (4 faces identiques)
- nombre de pokers (5 faces identiques).

Bien entendu, lors d'un lancer, on ne comptabilisera que le résultat le plus favorable. Ainsi, l'obtention de 4 faces identiques sera comptée comme un carré, et dès lors non comme un brelan, ni non plus comme une double paire ou une paire ! On ne demande pas de détecter et de comptabiliser les grandes suites ni les petites suites.

2.2.3 Les sorties

Un premier écran d'avertissement, comme celui qui suit, précisera l'objectif du programme à l'utilisateur.

```
Je vais lancer pour vous 1000 fois 5 dés.
Je vous afficherai (sous forme triée) chacun des
lancers obtenus.

Ensuite, je vous fournirai le nombre de paires,
de doubles paires, de brelans, de fulls, de carrés,
et de pokers obtenus au cours de ces 1000 lancers.




 Appuyez sur Entrée pour poursuivre
```

Ensuite les divers lancers obtenus seront affichés à raison de 10 par lignes et de 20 lignes par écran. Un lancer sera affiché avec les diverses faces obtenues rangées dans l'ordre croissant. Ainsi on notera :

1 1 3 3 4 et non 1 3 1 4 3 ou 3 1 4 1 3 ...

Les divers écrans proposés seront donc du genre suivant :

```
44456  12566  12235  13356  22344  33446  23344  13456  13455  12246
14456  12256  12344  22234  12346  12224  22344  34445  13566  12456
44556  33344  11345  23566  12555  23344  25566  11446  14556  24566
11135  12355  12345  12336  24456  44566  13455  22556  11156  23345
11345  34446  12256  22235  23344  12466  24566  13345  11246  11123
12356  23566  33456  13355  12233  13355  12366  14556  12346  12345
14566  11335  12445  24566  22246  14456  24455  22356  23356  11135
12444  22335  11356  12335  34566  12234  24566  13456  35556  12366
12346  13445  11334  23456  12355  22456  13345  22255  13344  11124
12346  12356  24556  11346  55666  23455  45566  12346  12236  23356
22366  11236  11346  33356  25566  24566  33356  12236  35556  22455
12256  11124  24446  13346  12344  44445  35566  23366  12335  22236
13446  12355  33355  12366  13444  11345  13556  22336  12566  11336
11256  12345  34566  12555  11223  44556  12356  23446  23556  11226
11116  23666  14556  23444  13556  11245  12466  11233  13455  12445
23456  23456  22226  22235  12234  34666  11356  13446  23466  11356
23466  33566  11366  15566  12466  12456  22345  34445  23356  13444
33556  22345  12346  25566  22456  23345  12345  33556  14466  12556
12244  23566  13566  12336  11456  13446  24556  13356  22236  11146
14555  12456  23345  12334  22223  16666  12245  14556  34466  11123
Appuyez sur la touche Entrée
```

les divers lancers présentés sur une même ligne étant séparés par 2 espaces.

A chaque fois qu'un écran sera rempli (par 20 lignes de 10 lancers) l'affichage (et le programme) s'interrompra jusqu'à ce que l'utilisateur presse la touche Entrée.

Enfin, à l'issue de la série d'écrans ainsi affichés (et représentant les 1000 lancers de 5 dés), un nouvel écran précisera les résultats obtenus, sous la forme suivante :

```
Voici les résultats des 1000 lancers :


Nombre de paires : 512

Nombre de doubles paires : 214

Nombre de brelans : 193

Nombre de fulls : 39

Nombre de carrés : 17

Nombre de pokers : 0

Appuyez sur Entrée
```

2.3 Jeu de poker : Comment faire ?

Ici aussi, et pour tenir compte de contraintes (dues à l'exécutant) qui apparaîtront en aval, je vous donne, non pas 5 dés mais un seul dé pour vous débrouiller. Pas question donc de lancer 5 dés d'un coup et de parcourir des yeux les 5 faces obtenues.

1. Il nous faudra effectuer l'équivalent de 1000 lancers. Il nous faudra donc à 1000 reprises effectuer le même travail.
2. Ce travail, ce sera de lancer à 5 reprises l'unique dé disponible (pour simuler les lancers simultanés des 5 dés que nous n'avons pas). Le premier petit problème est de décider sous quelle forme nous allons retenir les résultats de ces 5 lancers successifs du dé. On pourrait par exemple noter à la suite l'une de l'autre les 5 faces successivement obtenues : par exemple

   1 2 6 1 4

   On pourrait aussi se préparer une courte liste

   1 2 3 4 5 6

   et à chacun des 5 lancers du dé, placer un petit trait sous le résultat obtenu.

   On aurait ainsi

1	2	3	4	5	6
II	I		I		I

   Cette seconde manière de procéder est peut être moins naturelle, mais beaucoup plus adaptée à l'affichage du résultat du quintuple lancer tel que les spécifications l'exigent :

   1 1 2 4 6

   et surtout semble permettre une détection beaucoup plus aisée du résultat au sens du poker : paire, double-paire ...

   La première manière de noter, chronologiquement en quelque sorte, les 5 faces obtenues exigerait un tri pour l'affichage et également pour le repérage du résultat (au sens du poker).

   Je décide donc de noter le résultat du lancer sous la forme d'un tableau

1	2	3	4	5	6
II	I		I		I

Je laisserai d'ailleurs à demeure les 6 numéros me contentant à chaque nouveau quintuple lancer d'effacer d'abord les traits du précédent.

3. Il me faut aussi, au fur et à mesure des quintuples lancers, détecter le résultat (au sens du Poker) obtenu et le comptabiliser. Le plus simple est probablement de noter, également à demeure, une liste :

   paire double-paire brelan full carré poker

   et de comptabiliser le résultat obtenu à chaque quintuple lancer (et visible sous la forme présentée ci-dessus) en plaçant un trait supplémentaire sous le résultat.

   Par exemple :

1	2	3	4	5	6
II	I		I		I

   conduirait à placer un trait supplémentaire sous paire :

paire	double-paire	brelan	full	carré	poker
II	I		II	I	

Ainsi donc une stratégie générale, qui fait apparaître la manière dont les données seront retenues, peut s'énoncer :

A 1000 reprises

- Lancer 5 fois le dé en notant les résultats obtenus par des traits sous une liste

1	2	3	4	5	6
II	I		II		I

  Attention à bien effacer à chaque fois les traits du tirage précédent
- Annoncer le tirage obtenu;
- Déduire de ce tirage le résultat correspondant (au poker).
- Sur base de ce résultat, placer un trait supplémentaire sous l'élément correspondant de la liste

paire	double-paire	brelan	full	carré	poker
II	I		II	I	

Annoncer enfin le nombre de résultats de chaque type.

2.4 Jeu de poker : Comment faire faire

2.4.1 Structures de données

2.4.1.1 Constantes

? Quelles sont, à votre avis, les constantes importantes intervenant dans la description de la tâche évoquée ?

Deux constantes essentielles doivent être retenues :

 © CeFIS - FUNDP 2002

- NL, le nombre de lancers demandés, égal à 1000.
- ND, le nombre de dés lancés à chaque essai, égal à 5.

On pourrait être tenté d'y adjoindre une troisième constante représentant le nombre de faces d'un dé. Je ne le ferai pas ici. En effet, si l'on peut admettre que la description de la tâche abordée ici nécessite la précision du **nombre total de lancers** et du **nombre de dés utilisés** à chaque lancer, constantes descriptives dont les valeurs pourraient être différentes de ce qu'elles sont, en revanche le **nombre de faces d'un dé** ne peut être que ce qu'il est, soit 6, et il est inutile dès lors d'insister sur une constante sur la valeur de laquelle tout le monde doit s'accorder.

> Ce choix est bien entendu discutable et rien n'empêche d'adjoindre aux constantes inhérentes à la description du problème une troisième constante NF (pour nombre de faces du dé). Mais jamais NF ne pourra être différent de 6 !
>
> L'important, on l'aura compris, c'est de ne pas omettre cette étape de réflexion à propos des constantes importantes intervenant dans la description de la tâche concernée.

On pourra cependant, puisque Pascal le permet, mettre clairement en évidence que les entiers de 1 à 6 sont particulièrement importants dans le problème traité en définissant un type Faces constitué des seuls entiers entre 1 et 6. (`type Faces=1..6`).

2.4.1.2 Tableaux

? Comment représenteriez-vous les résultats du quintuple lancer d'un dé ?

La définition d'un premier tableau permettant de rendre compte de la liste et des traits représentant le résultat d'un quintuple lancer est immédiate : il nous faut un tableau à 6 composantes entières. Comme nous souhaitons que la composante d'indice N (= le tiroir n° N) contienne le nombre de dés (sur les cinq) qui montrent la face N, nous choisirons tout naturellement comme indices (= comme étiquettes des tiroirs) les entiers de 1 à 6 (ceux que nous avons baptisés comme étant du type Faces). Il ne reste plus qu'à nommer ce tableau : nous l'appellerons tout naturellement *Lancer* et nous pouvons l'imaginer comme :

Lancer

1	2	3	4	5	6
2	1	0	1	0	1
↑	↑	↑	↑	↑	↑
nombre de dés montrant 1	nombre de dés montrant 2	nombre de dés montrant 3	nombre de dés montrant 4	nombre de dés montrant 5	nombre de dés montrant 6

à chaque quintuple lancer

Le problème est plus délicat en ce qui concerne le second tableau à définir. La solution retenue sera d'ailleurs tributaire d'un apport d'informations nouvelles à propos des possibilités du langage Pascal. L'idée dont il faut arriver à rendre compte est celle d'une configuration :

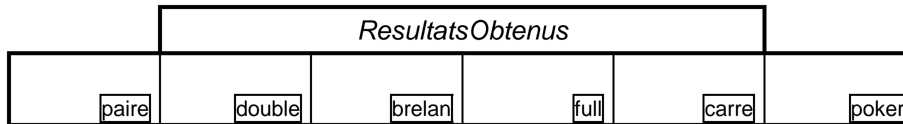
paire	double-paire	brelan	full	carré	poker
II	I		II	I	

On devine que l'idéal serait un tableau d'entiers, la première composante (= le premier tiroir) contenant le nombre de paires, la seconde le nombre de doubles-paires, etc.

? Quel type d'étiquettes choisiriez-vous pour le tableau qui enregistrera les résultats ?

On sent fort bien également que si l'on souhaite rester proche des termes mêmes du problème, les étiquettes retenues devraient être : "paire", "double-paire", "brelan", "full", "carré" et "poker" et non 1, 2, 3, 4, 5, 6, qui seraient des notations qui n'ont rien à faire avec les informations telles que nous les nommons.

Il nous faudrait donc un tableau



Nous savons aussi que jusqu'ici les informations susceptibles de fournir les étiquettes d'un tableau doivent être de type scalaire : entier, caractère, booléen ou d'un type intervalle basé sur ces derniers. Il est en tout cas interdit que des chaînes de caractères, comme 'paire', 'double', 'brelan', ... puissent servir d'étiquettes.

Nous pourrions bien entendu coder les termes "paire", "double", "brelan", "full", "carré", "poker", par des caractères mnémoniques 'p', 'd', 'b', 'f', 'c', 'P'. Malheureusement, ces caractères dont la suite ne constitue pas un intervalle du type caractère ne peuvent pas non plus servir à l'étiquetage. C'est ici qu'une des possibilités offertes par Pascal va nous faciliter l'écriture

Il nous est en effet loisible de définir un type d'informations manipulables en **énumérant les constantes** (qui doivent forcément - comme toujours en informatique - être en nombre fini) constituant ce nouveau type.

Ici par exemple, nous définirons un type **énuméré** :

ResultatsPossibles = (rien, paire, double, brelan, full, carre, poker)

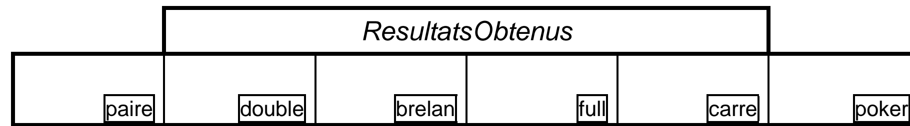
en Pascal :

```
type ResultatsPossibles = (rien, paire, double, brelan, full, carre, poker)
```

Je donnerai, à la suite du programme Pascal, quelques contraintes auxquelles sera soumise cette possibilité de type **énuméré**. Dès à présent, il est bon de signaler que **un type énuméré est un type scalaire** :

- L'ordre d'énumération est important; c'est dans cet ordre que les éléments constituant, par exemple, les étiquettes d'un tableau seront retenus.
- On pourra également écrire des boucles Pour... dont le compteur associé sera d'un type énuméré; ici aussi, on se contentera de citer la première et la dernière des valeurs prises par le compteur : l'ordre d'énumération fournira la succession des valeurs prises par ce dernier lors de la boucle.
- En d'autres termes, deux fonctions centrales permettant à un type de données de fournir les étiquettes d'un tableau ou encore un compteur de boucle Pour... existent pour un type énuméré : ce sont pred( ) et succ( ) **qui caractérisent les types scalaires**. Ajoutons encore que la fonction ord( ) existe aussi, le premier élément de l'énumération ayant le rang 0, le second le rang 1,etc.. (Cf. [Armici 86 A]., page 112).

Sans en dire davantage, nous devinons que cette possibilité constitue une aubaine pour le problème actuellement traité : après avoir défini le type énuméré ResultatsPossibles (comme ci-dessus) il nous reste seulement à demander un tableau :



L'expression en Pascal sera simplement :

```
ResultatsObtenus : array[paire..poker] of integer
```

Avec ces deux tableaux, nous venons de déterminer la structure que prendront les données. Il nous est à présent possible de décrire l'algorithme qui les fera utiliser.

2.4.2 Jeu de Poker : marche à suivre de premier niveau

AVERTIR
INITIALISER_RESULTATS
Pour *Compteur* allant de 1 à NL — *Compteur* entier entre 1 et NL
    FAIRE_UN_LANCER
    AFFICHER_LANCER
    EVALUER_RESULTAT — *Resultat* de type ResultatsPossibles
    Si *Resultat* <> rien alors
        ResultatsObtenus[*Resultat*] ← succ(*ResultatsObtenus*[*Resultat*])
AFFICHER_RESULTATS

Bien entendu, chacune des actions complexes qui y apparaît doit être soigneusement décrite (mais cela, nous commençons à en prendre l'habitude).

INITIALISER_RESULTATS

Placer 0 dans toutes les composantes du tableau *ResultatsObtenus*. → *ResultatsObtenus* (cf. ci-dessus)

avec

Avant INITIALISER_RESULTATS	Après INITIALISER_RESULTATS
- *ResultatsObtenus* tableau d'entiers d'indices paire à poker (intervalle du type énuméré ResultatsPossibles).	- Les composantes d'indices variant entre paire et poker de *ResultatsObtenus* sont toutes nulles.

FAIRE_UN_LANCER

Effectuer un quintuple lancer d'un dé et placer dans le tiroir n° i de *Lancer* le nombre de fois que la face i est apparue. → *Lancer* (cf. plus haut)

avec

Avant FAIRE_UN_LANCER	Après FAIRE_UN_LANCER
- *Lancer*, tableau d'entiers indexé par les entiers entre 1 et 6 (donc étiquettes de type Faces)	- Le ième tiroir de *Lancer* contient le nombre de fois que la face i est apparue sur les ND dés lancés

	AFFICHER_LANCER
Compteur → de type entier *Lancer* → tableau d'entiers indexé par les entiers entre 1 et 6 (donc étiquettes de type Faces)	Afficher dans l'ordre croissant des faces obtenues le résultat du quintuple lancer tel qu'il est représenté dans *Lancer*. Chaque affichage (qui comportera donc une suite ordonnée de ND nombres entre 1 et 6) sera séparé du suivant par deux espaces. De plus, sur base de la valeur de *Compteur* (qui donne le nombre de lancers déjà effectués), passer à la ligne tous les 10 lancers affichés et interrompre (jusqu'à ce que l'utilisateur frappe Entrée) quand 20 lignes sont affichées (ou après le dernier lancer).

avec

Avant AFFICHER_LANCER	Après AFFICHER_LANCER
- *Compteur* est un entier compris entre 1 et NL - Le ième tiroir de *Lancer* (tableau d'entiers indexé par les entiers entre 1 et 6 (donc étiquettes de type Faces)) contient le nombre de fois que la face i est apparue sur les ND dés lancés au quintuple lancer précédent	- *Compteur* n'a pas changé - *Lancer* n'a pas changé.

	EVALUER_RESULTAT	
Lancer → tableau d'entiers indexé par les entiers entre 1 et 6	Sur base du résultat précédent du lancer des ND dés, contenu dans *Lancer*, évaluer s'il s'agit d'une paire, d'une double paire, d'un brelan, etc. et placer la valeur détectée dans *Resultat*.	→ *Resultat* de type Resultats Possibles

avec

Avant EVALUER_RESULTAT	Après EVALUER_RESULTAT
- Le ième tiroir de *Lancer* contient le nombre de fois que la face i est apparue sur les ND dés lancés au quintuple lancer précédent	- *Lancer* n'a pas changé - *Resultat* contient l'une des valeurs : rien, paire, double, brelan, full, carre ou poker

	AFFICHER_RESULTATS
ResultatsObtenus→ (cf. plus haut)	Sur base des composantes de *ResultatsObtenus*, elle fait afficher le nombre de paires, de double paires, ... selon les spécifications de la page 95

avec

Avant AFFICHER_RESULTATS	Après AFFICHER_RESULTATS
- Les composantes d'indice paire, double, brelan,... de *Resultats-Obtenus* (tableau d'entiers d'indices paire à poker (intervalle du type énuméré ResultatsPossibles)) contiennent le nombre de paires, de doubles paires, de brelans, ...	Rien n'est modifié.

Cette première analyse nous permet dès à présent d'écrire la partie principale du programme Pascal :

2.5 *Jeu de Poker : Comment dire (1er niveau) ?*

```
program SIMULATION_POKER;
       (* Il fait simuler 1000 lancers de 5 dés (dont les faces portent comme il se doit les numéros
       de 1 à 6) fait afficher (sous forme triée) le résultat de chaque lancer et fait ensuite
       afficher le nombre de paires, de doubles-paires, de brelans, ... obtenus *)

uses WinCRT; (* pour les procédures de gestion d'écran *)
```

```
const  NL = 1000;
       (* NL est le nombre de lancers prévus *)
       ND = 5;
       (* ND est le nombre de dés jetés à chaque lancer *)

type   ResultatsPossibles = (rien,paire,double,brelan, full, carre, poker);           (1)
       (* on définit un type énuméré en donnant les constantes qui vont le constituer. Il s'agit ici
       des résultats possibles (au sens du poker) suite à un lancer *)
       Faces = 1..6;
       (* les nombres entiers entre 1 et 6 sont essentiels pour notre problème *)

var    Lancer : array[Faces] of integer;
       (* à chaque quintuple lancer, le tiroir 1 contiendra le nombre de dés (parmi les 5 lancés)
       montrant une face portant 1; le tiroir 2, le nombre de faces 2, ... *)
       ResultatsObtenus : array[paire..poker] of integer;                             (2)
       (* le tiroir d'étiquette paire contiendra le nombre de paires obtenues, celui d'étiquette
       double, le nombre de doubles paires, etc. Ces tiroirs sont ajustés au fur et à mesure des
       lancers successifs. *)
       Resultat : ResultatsPossibles;
       (* qui après chaque lancer contiendra le résultat obtenu (paire, double paire, brelan, ...)
       *)
       Compteur : integer;
       (* pour la boucle des NL lancers *)

procedure AVERTIR;
       (* elle fait effacer l'écran, fait afficher un message d'avertissement (voir les
       spécifications à propos des sorties) et lorsque l'utilisateur presse la touche Entrée fait à
       nouveau effacer l'écran. *)

procedure INITIALISER_RESULTATS;
       (* elle fait placer 0 dans les diverses composantes du tableau ResultatsObtenus ces tiroirs
       étant amenés ensuite à contenir le nombre de paires, de doubles paires,...*)

procedure FAIRE_UN_LANCER;
       (* elle fait simuler le lancer de ND (ici 5) dés et noter le nombre de 1 obtenus dans le
       tiroir n° 1 de Lancer, le nombre de 2 dans le tiroir n° 2, etc. *)

procedure AFFICHER_LANCER;
       (* elle fait afficher sous la forme demandée le quintuple lancer (représenté dans le tableau
       Lancer) qui vient d'être effectué : 10 lancers par ligne (séparés par 2 espaces), arrêt à 20
       lignes par écran, effacement lorsque l'utilisateur presse la touche Entrée. Le décompte des
       tirages effectués pour permettre cette présentation se fait par consultation de la variable
       Compteur *)

procedure EVALUER_RESULTAT;
       (* sur base du tableau Lancer (dont le tiroir n° I contient le nombre de sorties de la face
       I), cette procédure fait détecter s'il s'agit d'une paire, d'une double-paire, ... place le
       résultat de cette évaluation dans la variable Resultat *)

procedure AFFICHER_RESULTATS;
       (* A l'issue des NL lancers, elle fait afficher le nombre de paires obtenues (et stocké dans
       la composante paire de ResultatsObtenus), puis le nombre de doubles paires, ... *)

begin
randomize;
AVERTIR;
INITIALISER_RESULTATS;
for Compteur := 1 to NL do
    begin
    FAIRE_UN_LANCER;
    AFFICHER_LANCER;
    EVALUER_RESULTAT;
    if Resultat <> rien then                                                          (3)
       ResultatsObtenus[Resultat]:= succ(ResultatsObtenus[Resultat]);
    end;
AFFICHER_RESULTATS;
end.
```

2.5.1 Commentaires

(1) On définit un type énuméré en citant les constantes qui le constituent. Ces diverses constantes (dont la dénomination obéit aux règles habituelles relatives aux identificateurs Pascal) sont séparées par des virgules et la liste est enclose dans des parenthèses.

Les fonctions Ord, Pred et Succ existent pour un tel type énuméré.

Il faut encore ajouter que

- on ne peut demander ni l'affichage ni la lecture des valeurs d'un type énuméré;
- une même constante ne peut être présente dans deux type énumérés différents;
- un type énuméré ne peut reprendre ni des entiers, ni des réels, ni aucune constante d'un autre type;
- on peut bien entendu définir des types intervalles sur base d'un type énuméré.

(2) Un intervalle, comme ici paire..poker, d'un type énuméré peut servir à la définition des indices d'un tableau (= des étiquettes d'une étagère).

(3) Il faut bien veiller à écrire sans apostrophes les constantes d'un type énuméré.

2.6 Jeu de poker : analyse de second (et troisième) niveaux

A un premier niveau d'analyse, le problème de programmation posé est résolu. Il reste cependant à analyser chacune des actions mises en évidence.

2.6.1 Analyse de FAIRE_UN_LANCER

Comme le montrent les spécifications de la page 99, il nous faut faire procéder à un quintuple lancer de dé et noter au fur et à mesure dans *Lancer* les résultats obtenus.

2.6.1.1 FAIRE_UN_LANCER ? Comment faire ?

Le seul détail à ne pas omettre est, avant l'enregistrement de chaque nouveau quintuple lancer, de remettre à 0 les 6 composantes du tableau *Lancer*.

Ensuite, il est bien simple de procéder à ND (5) lancers successifs, en incrémentant de 1 à chaque fois le tiroir de *Lancer* dont l'étiquette correspond à la face obtenue.

2.6.1.2 FAIRE_UN_LANCER ? Comment faire faire ?

Pour *Face* allant de 1 à 6 — *Face* de type Faces
    Place 0 dans le tiroir *Face* de *Lancer*
Pour *Jet* allant de 1 à ND — *Jet* de type 1..ND
    Place dans *FaceObtenue* un nombre au hasard entre 1 et 6 — *FaceObtenue* de type Faces
    Augmente de 1 le tiroir n° *FaceObtenue* de *Lancer*

La traduction en Pascal est immédiate :

2.6.1.3 FAIRE_UN_LANCER ? Comment dire ?

```
procedure FAIRE_UN_LANCER;
       (* elle fait simuler le lancer de ND (ici 5) dés et noter le nombre de 1 obtenus dans le
       tiroir n° 1 de Lancer, le nombre de 2 dans le tiroir n° 2, etc. *)

var    Jet : 1..ND;
       (* pour la boucle décrivant les ND (5) lancers d'un dé *)
       FaceObtenue,
       (* résultat du lancer d'un dé *)
```

```
       Face : Faces; (* Faces est l'intervalle des entiers de 1 à 6 *)
       (* pour la boucle de mise à 0 des tiroirs de Lancer avant un nouveau lancer de ND dés *)

begin
for Face :=1 to 6 do                                                        (1)
    Lancer[Face]:=0;
for Jet := 1 to ND do
    begin
    FaceObtenue := random(6)+1;
    Lancer[FaceObtenue]:=Lancer[Faceobtenue]+1;
    end;
end;
```

2.6.1.4 Commentaires

Il faut remarquer que *Face* est un compteur de type Faces, donc astreint à être compris entre 1 et 6. Si nous retournons à la définition de la boucle For..., il nous faut écrire en **(1)** :

```
for Face :=1 to 6 do
    Lancer[Face]:=0;
```

⇨

```
Face := 1;
while Face <= 6 do
      begin
      Lancer[Face]:=0;
      Face := succ(Face);
      end;
```

Il y a pourtant une différence entre ces deux manières d'exprimer la répétition. Dans le cas du while..., à la fin de la boucle, le compteur *Face* finit par valoir 7, ce qui provoque une erreur puisque *Face* est astreint à rester entre 1 et 6, puisqu'il est de type Faces.

Avec la boucle For..., le fait que la variable *Face* soit astreinte à être comprise entre 1 et 6 est tout à fait compatible avec l'écriture d'une boucle For... allant de 1 à 6.

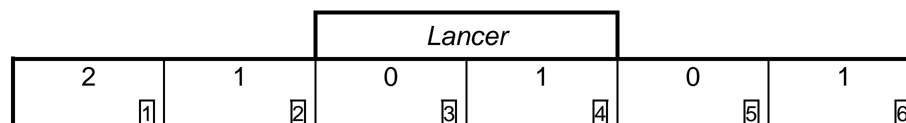
2.6.2 Analyse de AFFICHER_LANCER

C'est ici que (comme fort souvent) les spécifications à propos des sorties sont les plus prégnantes. Deux problèmes doivent être abordés :

1. Sur base du contenu de *Lancer*, afficher de manière ordonnée, les ND faces obtenues, en répétant autant de fois qu'il faut les faces qui se sont présentées plusieurs fois.
2. Les affichages doivent se faire à raison de 10 par ligne et de 20 lignes par écran.

2.6.2.1 AFFICHER_LANCER ? Comment faire ?

1. Nous disposons d'une description d'un quintuple lancer sous la forme :



L'affichage des résultats se fera par un parcours des 6 tiroirs. Pour chacun des tiroirs de contenu non nul une nouvelle boucle sera effectuée pour afficher l'étiquette portée par le tiroir autant de fois que cela est indiqué par le contenu de ce tiroir.

2. Les affichages se feront sans passage à la ligne, mais tous les 10 affichages on fera un tel passage à la ligne et tous les 200 affichages (ou lorsque le nombre d'affichages -donc de quintuple lancer- sera égal à NL (1000)) on affichera un message invitant l'utilisateur à appuyer sur la touche Entrée et on commandera une lecture (par readln seul) pour interrompre l'exécution.

   C'est à chaque fois la valeur de *Compteur,* fournissant le n° du quintuple lancer, qui le permettra : lorsque *Compteur* est multiple de 10 on commande un passage à la ligne, lorsqu'il est multiple de 200 ou égal à NL on commande un arrêt-lecture.

? Comment traduiriez-vous en une marche à suivre cette stratégie ? Comment détecter que *Compteur* est multiple de 10 ou de 200 ?

2.6.2.2 AFFICHER_LANCER ? Comment faire faire ?

L'analyse de ce petit travail n'est pas simple : il importe de bien saisir le rôle des 2 boucles Pour... imbriquées : la première fait parcourir successivement les divers tiroirs de *Lancer*, et pour chaque tiroir de contenu non nul, la seconde fait afficher autant de fois l'étiquette du tiroir concerné qu'il y a d'unités contenues dans ce tiroir.

```
Pour C allant de 1 à 6                              C de type Faces
    Pour CC allant de 1 à Lancer[C]                 CC de type intervalle entier 0..ND(1)
        Affiche C (sans passage à la ligne)
Affiche deux espaces
Si Compteur est multiple de 10 alors                (2)
    Passe à la ligne
Si Compteur multiple de 200 ou Compteur=NL alors    (2)
    Lecture-arrêt et efface l'écran
```

Deux remarques s'imposent :

(1) On pourrait s'étonner de ce que la boucle qui commande l'affichage répété d'une des faces obtenues ne soit pas précédée d'un test sur le fait que le contenu du tiroir correspondant n'est pas nul. En effet, la boucle est toujours faite de 1 à *Lancer*[*C*] sans tester cette valeur *Lancer*[*C*]. Il suffit de se rappeler que la boucle Pour... se comporte comme une structure Tant que... et que dès lors, lorsque *Lancer*[*C*] est nul le corps de la boucle (de 1 à 0) n'est pas effectué du tout.

On pourrait avec raison s'étonner du fait que *CC* est une variable de type intervalle de **0** à ND (et non de **1** à ND). En effet, *CC*, compteur de boucle varie de 1 à *Lancer*[*C*]. Comme *Lancer*[*C*] peut être nul, on est forcé d'inclure la valeur 0 dans les valeurs possibles de *CC*.

Il faut donc reconnaître que l'identification de la boucle Pour... à une boucle Tant que... particulière présente quelques ratés. En effet :

- Nous savons déjà qu'à la sortie d'une boucle Pour..., la valeur du compteur de boucle est indéterminée; ce ne serait pas le cas avec une boucle Tant que...
- Si la variable compteur d'une boucle est définie dans l'intervalle vi..vf, la boucle Pour... peut se dérouler de vi à vf. Avec une boucle Tant que..., on aurait une erreur car le compteur finirait avec la valeur succ(vf) et non vf.
- Et enfin, ici, nous sommes forcé d'inclure dans l'intervalle des valeurs possibles de la variable compteur d'une boucle Pour... la valeur terminale, alors que ceci ne devrait pas être le cas avec une expression faisant usage du Tant que...

Ainsi, au lieu de

```
Pour C allant de 1 à 6                              C de type Faces
    Pour CC allant de 1 à Lancer[C]                 CC de type intervalle entier 0..ND
        Affiche C (sans passage à la ligne)
```

on pourrait écrire :

```
Pour C allant de 1 à 6                              C de type Faces
    CC ← 1
    Tant que CC ≤ Lancer[C]                         CC de type intervalle entier 1..ND
        Affiche C (sans passage à la ligne)
        CC ← succ(CC)
```

sans que des problèmes se posent.

Face à ces problèmes relatifs à la définition de variables de type intervalle (comme ici *C* ou *CC*) et leurs rapports avec la boucle For..., on peut réagir de deux manières :

- On définit plus largement les variables sans les restreindre à un intervalle; ainsi ici, C et CC pourraient être simplement de type integer (ou byte ou word); on perd alors le bénéfice de mettre en évidence certains intervalles importants dans le problème abordé.
- Il est possible de demander, lors de la compilation (traduction) du programme Pascal que la version compilée du programme ne fasse aucune vérification du fait que les contenus des variables de type intervalle restent bien dans les bornes de ces intervalles. Dans ce cas, aucune erreur n'est détectée lorsqu'un contenu de variable sort des bornes fixées à cette variable.

(2) C'est évidemment l'opération *Compteur* mod 10 (ou mod 200) qui en donnant le reste de la division de *Compteur* par 10 (ou 200) permet de tester si *Compteur* est multiple de 10 (ou 200).

Nous sommes en mesure de traduire :

2.6.2.3 AFFICHER_LANCER ? Comment dire ?

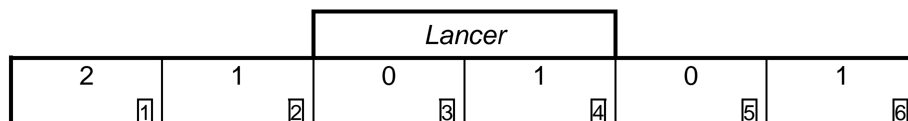
```
procedure AFFICHER_LANCER;
        (* elle fait afficher sous la forme demandée le quintuple lancer (représenté dans le tableau
        Lancer) qui vient d'être effectué : 10 lancers par ligne (séparés par 2 espaces), arrêt à 20
        lignes par écran, effacement lorsque l'utilisateur presse la touche Entrée. Le décompte des
        tirages effectués pour permettre cette présentation se fait par consultation de la variable
        Compteur *)
var     C : Faces;
        (* pour la boucle de parcours des 6 tiroirs de Lancer *)
        CC : 0..ND;
        (* pour la boucle qui fera imprimer chaque numéro le nombre de fois qu'il avait été tiré *)

begin
for C := 1 to 6 do
        (* affichage sous forme triée d'un lancer *)
        for CC:= 1 to Lancer[C] do
               write(C);
write ('  '); (* affichage, sans changer de ligne de deux espaces à la suite de la composition du
        lancer *)
(* tous les 10 lancers, passage à la ligne *)
if Compteur mod 10 = 0 then
   writeln;
(* tous les 200 lancers ou lorsque le nombre de lancers est celui demandé au total arrêt puis
        effacement *)
if (Compteur mod 200 = 0) or (Compteur = NL) then
   begin
   writeln;
   write('Appuyez sur la touche Entrée ');
   readln;
   clrscr;
   end;
end;
```

2.6.3 Analyse de EVALUER_RESULTAT

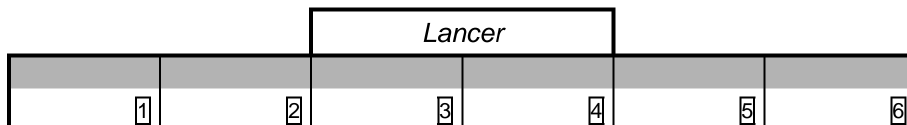
2.6.3.1 EVALUER_RESULTAT ? Comment faire ?

Voici sans doute la partie la plus intéressante. Disposant du tableau *Lancer*, il nous faut déterminer si nous avons obtenu une paire, une double paire,... (Cf. p 100).

Bien entendu, comme toujours avec les tableaux, notre stratégie doit s'imposer la contrainte que les contenus des tiroirs sont masqués. Sinon, "on voit" qu'un tableau comme



"montre" clairement une double paire. Il nous faudra donc raisonner sur un tableau



? Comment détecter que le contenu de *Lancer* donne une paire, une double paire,... ?

Il est cependant facile de voir qu'il suffit de connaître le plus grand, ou, dans certains cas, les deux plus grands nombres contenus dans les tiroirs de *Lancer* pour déterminer le résultat obtenu.

Rappelons que le tiroir n° N de *Lancer* contient le nombre de N obtenus lors du quintuple lancer.

Si le plus grand nombre contenu dans les tiroirs de *Lancer* est 5, nous avons bien entendu un poker, si c'est 4, un carré. Si c'est 3 et si le suivant est 2, c'est un full et si le suivant n'est pas 2, c'est d'un simple brelan qu'il s'agit. Enfin lorsque le plus grand et le suivant sont égaux à 2, c'est une double paire; et une paire lorsque le plus grand est 2 et le suivant n'est pas 2. Et pour terminer, si le plus grand n'est pas 2, nous n'obtenons rien.

Nous voici en mesure d'écrire

2.6.3.2 EVALUER_RESULTAT ? Comment faire faire ?

C'est bien entendu une cascade d'alternatives qui va nous être nécessaire :

```
DETERMINER le plus grand élément et le suivant dans Lancer          PlusGrand,
Si PlusGrand = 5 alors                                              Suivant : de type entier
        Resultat ← poker
sinon
        Si PlusGrand = 4 alors
                Resultat ← carre
        sinon
                Si PlusGrand = 3 alors
                        Si Suivant = 2 alors
                                Resultat ← full
                        sinon
                                Resultat ← brelan
                sinon
                        Si PlusGrand = 2 alors
                                Si Suivant = 2 alors
                                        Resultat ← double
                                sinon
                                        Resultat ← paire
                        sinon
                                Resultat ← rien
```

Nous avons de la sorte fait apparaître la nécessité de deux variables : *PlusGrand* et *Suivant* qui devront renfermer les deux plus grands contenus des tiroirs de *Lancer* (le maximum et son suivant, éventuellement égal à ce maximum). Ce sont des entiers (en réalité compris entre 0 et ND (=5)).

Mais il nous reste le problème de la détermination de ces deux plus grands éléments de *Lancer* :

 © CeFIS - FUNDP 2002

<table>
<tr><td></td><td colspan="1">DETERMINER_PLUS_GRAND_ET_SUIVANT</td><td></td></tr>
<tr><td>Lancer →
tableau d'entiers indexé par les entiers entre 1 et 6 (donc étiquettes de type Faces)</td><td>Déterminer les deux plus grandes composantes de Lancer (éventuellement égales) et les placer dans PlusGrand (pour la plus grande) et dans Suivant (pour celle qui suit).</td><td>→ PlusGrand
de type entier
→ Suivant
de type entier</td></tr>
</table>

avec

Avant DETERMINER...	Après DETERMINER...
- Les tiroirs de *Lancer* contiennent des entiers entre 0 et ND (5), la somme des contenus de ces tiroirs faisant ND.	- *Lancer* n'a pas changé - *PlusGrand* contient le plus grand élément de *Lancer* - *Suivant* contient le plus grand élément de *Lancer* privé de *PlusGrand*

Notons bien qu'il ne s'agit pas ici du problème général (et isolé) de déterminer les deux éléments maximaux d'un tableau, mais de placer dans les deux variables *PlusGrand* et *Suivant* les deux éléments maximaux du tableau *Lancer*.

Cette remarque prendra toute son importance lorsqu'il s'agira de montrer les limites d'une démarche descendante stricte.

Nous sommes cependant déjà en mesure de préciser :

2.6.3.3 EVALUER_RESULTAT ? Comment dire ?

```
procedure EVALUER_RESULTAT;
       (* sur base du tableau Lancer (dont le tiroir n° I contient le nombre de sorties de la face
       I), cette procédure fait détecter s'il s'agit d'une paire, d'une double-paire, ... et place
       le résultat de cette évaluation dans la variable Resultat *)
var    PlusGrand, (* plus grand élément de Lancer *)
       Suivant (* second élément en taille de Lancer *)
       : integer; (* on pourrait les déclarer entre 0 et ND (5) *)

procedure DETERMINER_PLUS_GRAND_ET_SUIVANT;
       (* elle fait calculer les deux éléments maximaux (qui peuvent être identiques) du tableau
       Lancer et les fait placer dans PlusGrand et Suivant *)

begin (* début de EVALUER_RESULTAT *)
DETERMINER_PLUS_GRAND_ET_SUIVANT;
if PlusGrand = 5 then
   Resultat:=poker
else
    if PlusGrand = 4 then
       Resultat := carre
    else
       if PlusGrand = 3 then
          if Suivant = 2 then
             Resultat := full
          else
              Resultat := brelan
       else
           if PlusGrand = 2 then
              if Suivant = 2 then
                 Resultat := double
              else
                  Resultat := paire
           else
               Resultat :=rien;
end;
```

Je me permettrai d'appeler dans la suite DETERMINER la procédure correspondant aux spécifications décrites ci-dessus (et notée DETERMINER_PLUS_GRAND_ET_SUIVANT).

Nous avons le choix pour la poursuite de l'effort en cours de laisser un moment de côté l'action DETERMINER pour continuer les analyses des autres procédures de "second niveau" AVERTIR, INITIALISER_RESULTATS et AFFICHER_ RESULTATS. Dans l'exploration de

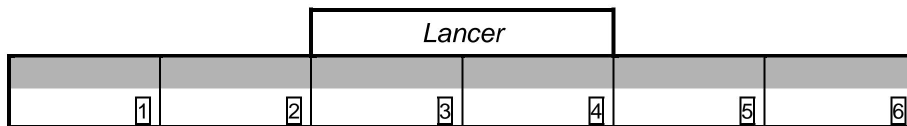
l'arborescence des procédures et sous-procédures, nous pouvons privilégier une stratégie par niveaux ou une stratégie qui descende aussi loin que possible le long d'une branche.

Comme les spécifications de DETERMINER sont encore toutes fraîches, nous nous intéresserons d'abord à cette dernière.

? Comment vous y prendriez-vous pour déterminer les deux éléments maximaux de *Lancer* ?

2.6.4 Analyse de DETERMINER... (le plus grand et le suivant dans *Lancer*)

Voilà un beau problème d'algorithmique "pure" et qui offre de nombreuses solutions. Nous sommes face à



et nous devons en déterminer les deux éléments maximaux.

2.6.4.1 DETERMINER... ? Comment faire ?

L'analyse menée et qui a conduit aux spécifications de DETERMINER nous impose de plus de ne pas modifier le tableau *Lancer*.

> Il est pourtant vrai qu'ici, si l'on reprend le décorticage du problème déjà effectué, on pourrait sans danger modifier le contenu de *Lancer*, puisque, si l'on est attentif, on verra que l'opération suivante qui lui sera appliquée consistera à placer 0 dans toutes ses composantes.
>
> On comprend aussi aisément qu'il est impératif de s'en tenir aux spécifications de l'action (DETERMINER) qui reste à analyser si l'on veut que l'approche descendante garde son sens.

La solution la moins "fatigante" (mais aussi la moins futée) pour trouver le plus grand élément et son suivant consiste à faire deux explorations successives de *Lancer* : à la première on cherche seulement le plus grand élément puis on fait un second passage après avoir fait en sorte que le plus grand déjà détecté ne soit plus candidat, en ayant pris soin de le "marquer" (par exemple en changeant son signe).

Attention après les deux passages de veiller à restaurer *Lancer* dans son état initial (puisque les spécifications de DETERMINER l'exigent).

Notons aussi que nous pourrions mettre en oeuvre des stratégies plus fines qui tiennent compte de la structure particulière du tableau *Lancer*. Ainsi, si pendant l'exploration, nous trouvons 5 ou 4 comme contenu d'un tiroir nous pourrions nous arrêter là, la valeur du suivant pouvant en être immédiatement déduite.

Il est une seconde stratégie, plus conforme aux principes d'une saine construction d'algorithme, qui fait en quelque sorte usage d'un principe de récurrence : on suppose le travail fait jusqu'à un certain point et on cherche comment faire un pas de plus. On a exploré *Lancer* jusqu'à un certain tiroir qu'on vient de refermer en ayant déterminé jusque là le plus grand et le suivant; on ouvre le tiroir qui suit, que faire ?

> Il s'agit bien entendu là d'une méthodologie de construction d'algorithme fort connue. Jacques ARSAC, qui est sans doute le chercheur qui a fait le plus pour vulgariser ces concepts en use constamment dans son dernier livre consacré à la programmation ("Préceptes pour programmer". Cf [Arsac 91] (page 231) Si vous aimez l'informatique et si vous comptez vous retirer seul dans une ile déserte avec un minimum de nourritures intellectuelles pour y survivre, un conseil : jetez votre ordinateur mais prenez avec vous cet ouvrage, il vous fera passer des moments passionnants.

Appelons donc si vous le voulez bien, pour ne pas nous laisser distraire par des indices (des étiquettes), *Nouveau* le contenu du tiroir qui vient d'être ouvert. Nous avons jusque là un *PlusGrand* et un *Suivant* lié par la relation (si le travail a jusque là été bien fait, mais cela nous l'avons supposé) *Suivant* ≤ *PlusGrand*.

Trois cas peuvent se présenter :

- *Nouveau* ≤ *Suivant* : rien ne doit alors être changé, le *PlusGrand* reste le plus grand et *Suivant* le suivant;
- *Nouveau* ≥ *PlusGrand* : alors *Nouveau* devient le nouveau *PlusGrand* et ce dernier devient le nouveau *Suivant*;
- *Suivant* < *Nouveau* < *PlusGrand* : alors le *PlusGrand* reste ce qu'il était, mais *Nouveau* devient le nouveau *Suivant*.

Le processus se termine bien entendu lorsque le *Nouveau* examiné était le contenu du dernier tiroir de *Lancer*. Il reste à initialiser le mouvement (puisque nous avons supposé que le travail avait déjà été entamé) en précisant ce qui a été fait avant la première ouverture d'un "nouveau" tiroir effectuée lors du processus décrit ci-dessus. En réalité, il est indispensable d'être certain qu'on commence le processus en étant déjà sûr d'avoir le *PlusGrand* et le *Suivant*. Ceux-ci sont obligatoirement les contenus des deux premiers tiroirs de *Lancer*, le plus grand des deux étant placé dans *PlusGrand* et le second (en taille) dans *Suivant*. C'est une simple comparaison qui permettra de décider lequel des deux premiers tiroirs deviendra *PlusGrand* et lequel sera *Suivant*.

Nous allons à présent traduire ces deux manières de procéder en deux marches à suivre distinctes.

2.6.4.2 DETERMINER ? Comment faire faire ?

La première stratégie s'écrit :

```
PlusGrand ← 0                                                    (1)
Pour Compteur allant de 1 à 6                  Compteur de type Faces
      Si Lancer[Compteur] > PlusGrand alors
            PlusGrand ← Lancer[Compteur]
            IndicePlusGrand ← Compteur  IndicePlusGrand de type Faces
Lancer[IndicePlusGrand] ← - Lancer[IndicePlusGrand]              (2)
Suivant ← 0
Pour Compteur allant de 1 à 6
      Si Lancer[Compteur] > Suivant alors
            Suivant ← Lancer[Compteur]
Lancer[IndicePlusGrand] ← - Lancer[IndicePlusGrand]              (3)
```

(1) Avant le parcours de *Lancer* pour la recherche du plus grand élément, parmi tous les tiroirs **à contenu positif ou nul**, il suffit d'initialiser *PlusGrand* à 0.

(2) On fait en sorte que l'élément le plus grand de *Lancer* détecté à ce premier parcours ne soit plus candidat au second parcours en le rendant négatif.

(3) Il faut bien entendu restaurer *Lancer* dans son état initial en rétablissant le signe positif du tiroir contenant le plus grand élément de *Lancer*.

Nous avons donc dû définir deux variables de type Faces, *Compteur* pour les deux parcours de *Lancer* et *IndicePlusGrand* pour retenir le n° du tiroir de *Lancer* contenant le plus grand élément.

Il nous reste à illustrer la seconde stratégie développée ci-dessus :

Si *Lancer*[1]>*Lancer*[2] alors
    PlusGrand ← *Lancer*[1]
    Suivant ← *Lancer*[2]
sinon
    PlusGrand ← *Lancer*[2]
    Suivant ← *Lancer*[1]
Pour *Compteur* allant de 3 à 6 — *Compteur* de type Faces
    Nouveau ← *Lancer*[*Compteur*] — *Nouveau* de type 0..ND (5)
    Si *Nouveau* ≥ *PlusGrand* alors
        Suivant ← *PlusGrand*
        PlusGrand ← *Nouveau*
    sinon
        Si *Nouveau* > *Suivant* alors
            Suivant ← *Nouveau*

Nous voici donc en mesure de préciser de deux manières d'écrire la procédure :

2.6.4.3 DETERMINER_PLUS_GRAND_ET_SUIVANT ? Comment dire ?

```
procedure DETERMINER_PLUS_GRAND_ET_SUIVANT;
       (* elle fait calculer les deux éléments maximaux (qui peuvent être identiques du tableau
       Lancer et les fait placer dans PlusGrand et Suivant *)
var    Compteur, (* pour la boucle de parcours de Lancer *)
       IndicePlusGrand (* pour stocker l'indice du plus grand élément *)
       : Faces;
begin
PlusGrand:=0;
for Compteur := 1 to 6 do
    if Lancer[Compteur]>PlusGrand then
       begin
       PlusGrand := Lancer[Compteur];
       IndicePlusGrand := Compteur;
       end;
Lancer[IndicePlusGrand]:= -Lancer[IndicePlusGrand];
Suivant:=0;
for Compteur := 1 to 6 do
    if Lancer[Compteur] > Suivant then
       Suivant := Lancer[Compteur];
Lancer[IndicePlusGrand]:= -Lancer[IndicePlusGrand];
end;
```

et également

```
procedure DETERMINER_PLUS_GRAND_ET_SUIVANT;
       (* elle fait calculer les deux éléments maximaux (qui peuvent être identiques du tableau
       Lancer et les fait placer dans PlusGrand et Suivant *)
var    Compteur : Faces;(* pour la boucle de parcours de Lancer *)
       Nouveau : 0..5;(*qui contiendra les contenus successifs des tiroirs de Lancer à explorer *)
begin
if Lancer[1] > Lancer[2] then
   begin
   PlusGrand:=Lancer[1];
   Suivant:=Lancer[2];
   end
else
   begin
   PlusGrand:=Lancer[2];
   Suivant:=Lancer[1];
   end;
for Compteur := 3 to 6 do
    begin
```

```
    Nouveau := Lancer[Compteur];
    if Nouveau>= PlusGrand then
       begin
       Suivant:=PlusGrand;
       PlusGrand:=Nouveau;
       end
    else
       if Nouveau>Suivant then
          Suivant:=Nouveau;
    end;
end;
```

2.6.5 Analyse de AVERTIR, INITIALISER_RESULTATS et AFFICHER_RESULTATS

Il resterait à fournir les marches à suivre correspondant aux trois actions AVERTIR, INITIALISER_RESULTATS et AFFICHER_RESULTATS. Aucune de ces trois analyses ne présente la moindre difficulté et je me contenterai donc de fournir les textes des procédures correspondantes en Pascal.

2.6.5.1 AVERTIR ? Comment dire ?

```
procedure AVERTIR;
      (* elle fait effacer l'écran, fait afficher un message d'avertissement (voir les
      spécifications à propos des sorties) et lorsque l'utilisateur presse la touche Entrée fait à
      nouveau effacer l'écran. *)
begin
clrscr;
writeln('Je vais lancer pour vous ',NL,' fois ',ND,' dés.');
writeln('Je vous afficherai (sous forme triée) chacun des ');
writeln('lancers obtenus.');
writeln;
writeln('Ensuite, je vous fournirai le nombre de paires, ');
writeln('de doubles paires, de brelans, de fulls, de carrés,');
writeln('et de pokers obtenus au cours de ces ',NL,' lancers.');
gotoxy(2,20);  (* pour aller en bas de l'écran *)
writeln('Appuyez sur Entrée pour poursuivre ');
readln; (* pour une lecture qui va interrompre jusqu'à l'appui sur Entrée le déroulement *)
clrscr;
end;
```

2.6.5.2 INITIALISER_RESULTATS ? Comment dire ?

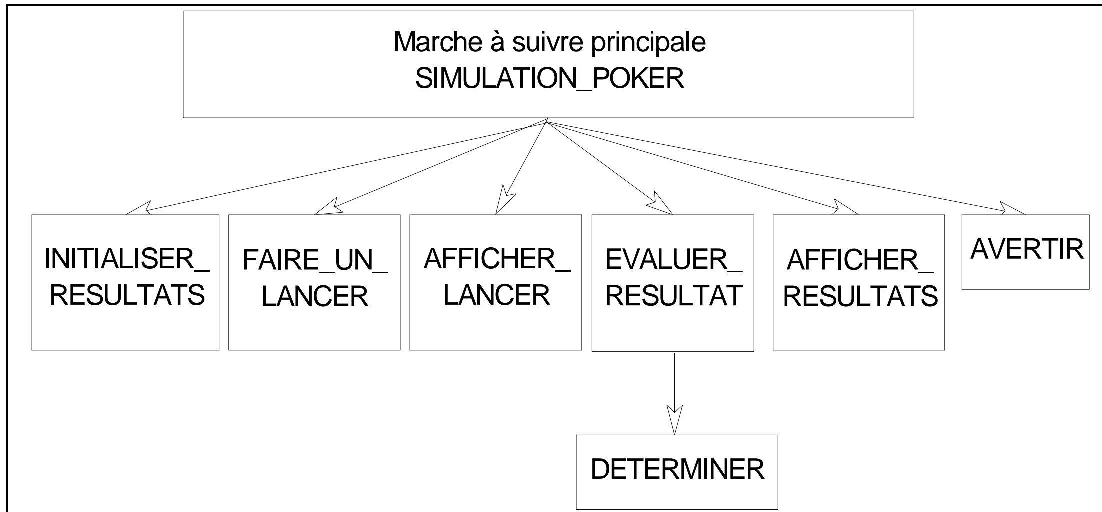
```
procedure INITIALISER_RESULTATS;
      (* elle fait placer 0 dans les diverses composantes du tableau ResultatsObtenus ces tiroirs
      étant amenés à contenir le nombre de paires, de doubles paires,...*)
var   Compteur : ResultatsPossibles;
      (* pour la boucle allant de paire à poker *)

begin
for Compteur := paire to poker do
    ResultatsObtenus[Compteur]:=0;
end;
```

2.6.5.3 AFFICHER_RESULTATS ? Comment dire ?

```
procedure AFFICHER_RESULTATS;
      (* A l'issue des NL lancers, elle fait afficher le nombre de paires obtenues (et stocké dans
      la composante paire de ResultatsObtenus), puis le nombre de doubles paires, ... *)
begin
writeln('Voici les résultats des ',NL,' lancers :');
writeln;
writeln;
writeln('Nombre de paires : ',ResultatsObtenus[paire]);
writeln;
writeln('Nombre de doubles paires : ',ResultatsObtenus [double]);
writeln;
writeln('Nombre de brelans : ',ResultatsObtenus[brelan]);
writeln;
writeln('Nombre de fulls : ',ResultatsObtenus[full]);
writeln;
writeln('Nombre de carrés : ',ResultatsObtenus[carre]);
writeln;
```

```
writeln('Nombre de pokers : ',ResultatsObtenus[poker]);
writeln;
writeln('Appuyez sur Entrée ');
readln;
end;
```

Il n'est plus nécessaire de redonner ici le texte de l'entièreté du programme. Je terminerai simplement en rappelant la structure générale :



3. Conclusions

Nous voici au terme de cet important chapitre : nous y avons découvert que pour programmer une tâche, il ne fallait pas aller d'emblée vers une longue écriture en terme d'actions élémentaires, mais plutôt vers un décorticage de la tâche globale en actions complexes, qui seront à leur tour, ensuite, décortiquées. Cette approche (descendante) permet, lorsqu'on s'intéresse à une des actions complexes apparues lors du décorticage, d'oublier le reste du problème pour se centrer sur l'analyse de cette action complexe particulière. Elle permet donc un travail modulaire.

Cette découpe du travail est rendue possible par le fait que chaque action complexe apparue lors du décorticage initial (analyse de premier niveau) est parfaitement décrite, de manière à ce que son traitement et son analyse soient possibles, sans connaître le reste du problème où elle s'insère.

Evidemment, cette approche peut sembler fort lourde pour l'instant, essentiellement parce que les tâches abordées restent relativement simples, conduisant à des programmes de petite taille. La spécification des actions complexes (procédures) est particulièrement lassante, puisque les marche à suivre résultantes sont souvent plus courtes que ces spécifications. Comme, de plus, nous avons ici l'ensemble du problème sous les yeux, nous n'ignorons pas ce qui se trouve "au-dessus" ou "à côté" du module en train d'être traité. Il n'est donc pas vrai ici que lors de l'ananlyse d'un module nous "oublions" le reste.

Enfin, il est souvent délicat pour les débutants de mettre en oeuvre la démarche descendante : pourquoi choisir tel décorticage et telles actions complexes plutôt que d'autres. Il est vrai que ces décorticages et les modules auxquels ils conduisent sont multiples : l'analyse de premier niveau de l'un ne sera pas celle d'un autre. Pourtant, les principes de cette démarche constituent des guides méthodologiques précieux que le débutant a intérêt à maîtriser et à mettre en oeuvre.

Cette approche descendante s'incarne parfaitement en Pascal à travers les procédures et l'usage des variables de différents niveaux. Nous verrons que, comprise trop strictement, elle

devient absurde : le chapitre suivant en montrera quelques limites. Mais ce n'est pas parce qu'il faut parfois s'en écarter, qu'il n'est pas intéressant de la mettre en pratique.

4. Exercices

1. Que pensez-vous des marches à suivre suivante en ce qui concerne l'action VERIFIER précisée dans la première analyse (page 75).

a.

```
Place 1 dans C
Tant que C< Compteur et non Egalite
    Place Tirages [C] = Tirages [Compteur] dans Egalite
    Place succ(C) dans C
```

b.

```
Pour C allant de 1 à pred(Compteur)
        Egalite ← Tirages [C] = Tirages [Compteur]
```

c.

```
Pour C allant de 1 à pred(Compteur)
        Si Tirages[C] = Tirages[Compteur] alors
                Egalite ← vrai
```

d.

```
Place 0 dans C
Répéter
        Place succ(C) dans C
        Si Tirages[C] = Tirages[Compteur] alors
                Egalite ← vrai
jusqu'à ce que C=pred(Compteur) ou Egalite
```

e.

```
Place 0 dans C
Répéter
        Place succ(C) dans C
jusqu'à ce que C=pred(Compteur) ou Tirages[C]=Tirages[Compteur]
Place Tirages[C]=Tirages[Compteur] dans Egalite
```

Pourriez-vous traduire en Pascal celles qui vous ont semblé correctes ?

2. Pourriez-vous programmer la seconde stratégie mise en évidence (page 67) à l'aide d'un tableau **entier** *DejaObtenus*.
3. Pourriez-vous écrire un programme qui simule 1000 tirages successifs du Lotto et indique ensuite le numéro "normal" sorti le plus fréquemment et celui sorti le moins souvent (ou la liste s'il y en a plusieurs). Donner les mêmes renseignements en ce qui concerne le numéro complémentaire.

Les divers tirages seront présentés à raison de 10 tirages par écran (avec arrêt après l'affichage de chacun d'eux jusqu'à ce que l'utilisateur presse une touche). Voici les deux premiers et le dernier écran présentant les tirages :

```
Je vais procéder pour vous à 1000 tirages
successifs du Lotto.
Je vous dirai ensuite quel numéro est sorti le
plus souvent parmi les numéros normaux et
lequel est sorti le moins souvent.
Je vous donnerai aussi les mêmes renseignements
pour la sortie du complémentaire.

Appuyez sur la touche Entrée.
```

```
Tirage 1
Numéros :18  3  11  25  41  28  Complémentaire :13
Tirage 2
Numéros :1  41  24  5  37  2  Complémentaire :12
Tirage 3
Numéros :20  36  9  41  37  5  Complémentaire :3
Tirage 4
Numéros :9  28  14  4  33  29  Complémentaire :27
Tirage 5
Numéros :10  14  21  32  6  8  Complémentaire :37
Tirage 6
Numéros :16  17  32  33  13  34  Complémentaire :28
Tirage 7
Numéros :23  22  37  29  5  39  Complémentaire :19
Tirage 8
Numéros :35  9  3  21  12  11  Complémentaire :39
Tirage 9
Numéros :39  21  32  4  37  41  Complémentaire :36
Tirage 10
Numéros :17  4  1  5  35  33  Complémentaire :32

Appuyez sur Entrée
```

```
Tirage 991
Numéros :26  15  22  37  5  34  Complémentaire :21
Tirage 992
Numéros :7  24  28  22  12  29  Complémentaire :6
Tirage 993
Numéros :6  15  40  32  25  12  Complémentaire :9
Tirage 994
Numéros :4  16  38  1  14  20  Complémentaire :9
Tirage 995
Numéros :29  16  5  12  10  13  Complémentaire :21
Tirage 996
Numéros :19  42  27  41  30  23  Complémentaire :32
Tirage 997
Numéros :38  27  34  4  25  3  Complémentaire :41
Tirage 998
Numéros :41  4  40  8  37  15  Complémentaire :12
Tirage 999
Numéros :9  1  39  26  18  21  Complémentaire :12
Tirage 1000
Numéros :4  15  35  30  41  34  Complémentaire :16

Appuyez sur Entrée
```

Le tout dernier écran affichera les renseignements globaux demandés sous la forme :

```
Numéro(s) le(s) plus fréquent(s) :
12  26
Numéro(s) le(s) moins fréquent(s) :
21
Complémentaire(s) le(s) plus fréquent(s) :
4  12  33
Complémentaire(s) le(s) moins fréquent(s) :
11  40
```

On veillera à adopter une approche descendante qui s'inspire d'une des stratégies développées plus haut et à réutiliser au maximum les actions complexes mises en évidence.

4.! Le traitement à automatiser (donc à programmer) est le suivant :

On souhaite fournir une somme (entière et strictement positive) représentant un montant en francs belges et obtenir sa décomposition en coupures utilisant le nombre minimal de ces dernières.

Les 8 coupures disponibles sont les billets de 2000 F, 1000 F, 500 F, 100 F et les pièces de 50 F, 20 F, 5 F et 1 F[11].

Voici quelques écrans explicitant les sorties attendues.

```
Vous allez me fournir une somme et je vous dirai combien de
billets et de pièces elle représente. Attention,
la somme ne peut depasser 32767 francs.
```

```
Quelle somme ? 27346
Cela fait
 13 billets de 2000 francs
  1 billet de 1000 francs
  3 billets de 100 francs
  2 pièces de 20 francs
  1 pièce de 5 francs
  1 pièce de 1 franc
On continue oui ou non ?
```

```
Quelle somme ? 4002
Cela fait
  2 billets de 2000 francs
  2 pièces de 1 franc
On continue oui ou non ?
```

On veillera à discuter l'influence du type de variable(s) choisi sur la taille du montant traitable par le programme.

[11] Vous pouvez y ajouter si vous le désirez les billets de 10000 F et de 200 F (ou poser le problème en Euros)...

3

Les limites de l'approche descendante

Même si la méthodologie mise en avant au chapitre précédent constitue l'un des guides essentiels, la réalité oblige quelquefois à transiger avec une approche descendante interprétée trop strictement.

Jusqu'ici, une action complexe, mise en évidence au cours de l'analyse, s'incarnait en une procédure qui sur base d'une liste de variables **préexistantes** consultables par cette procédure (celles notées à gauche du rectangle où l'on décrivait les spécifications), modifiait d'autres variables (ou les mêmes) **préexistantes** (celles notées à droite des mêmes rectangles).

Disposant des spécifications de cette action complexe (= procédure), qui comportait, bien entendu, les listes des variables consultables et modifiables par cette action, on pouvait, lors de l'analyse correspondante, oublier **tout** le reste.

> J'insiste ici essentiellement sur les variables : c'est à travers elles, en quelque sorte, que l'action complexe est liée au reste de la marche à suivre. Mais, nous le savons, c'est non seulement un certain nombre de variables préexistantes qui lie cette action au reste, mais également le fait que toutes les déclarations (de constantes et de types) préalables sont aussi accessibles. Ceci est particulièrement bien illustré au chapitre précédent dans la description métaphorique de la manière dont se passe l'appel de procédure.

Comme nous allons le voir, il arrive que diverses actions complexes mises en évidence au cours de l'analyse descendante, soient extrêmement proches l'une de l'autre : elles font faire le même travail par les exécutants qui en sont chargés; seules changent à chaque fois les listes des variables consultables et modifiables. En d'autres termes, ces procédures "jumelles" font faire la même chose, mais à propos de variables préexistantes différentes.

On devine qu'il serait ridicule, dans ce cas, en restant trop catégoriquement attaché à une approche descendante, de ne pas voir qu'on sera amené à analyser et à écrire plusieurs fois la "même" procédure : tout est pareil sauf les noms des variables auxquelles s'appliquent ces procédures. C'est donc ici que, pour la première fois, va intervenir la possibilité d'assortir une procédure d'un ensemble de paramètres, ce qui évitera des duplications inutiles.

C'est un travail d'une utilité toute relative, mais qui réjouira les amateurs d'histoire des nombres, qui va nous permettre de cerner ces limites de l'approche descendante : la confection d'une mini-calculatrice qui aurait réjoui Jules César, qui, comme chacun sait, notait les nombres en chiffres "romains"...

1. Une calculatrice pour le grand Jules

1.1 Description floue

En gros, ce qui est demandé, c'est de rendre l'exécutant-ordinateur capable d'additionner, de soustraire et de multiplier des nombres écrits en chiffres romains.

1.2 Quoi faire (faire) ?

Comme nous en avons pris l'habitude, voici quelques écrans successifs qui montrent de manière concrète ce qui est attendu :

```
Vous allez me fournir 2 nombres en chiffres romains et je vous
fournirai, toujours en chiffres romains, leur somme, leur
différence et leur produit.
ATTENTION, LES DEUX NOMBRES QUE VOUS ME DONNEZ DOIVENT ETRE
CORRECTEMENT ECRITS EN CHIFFRES ROMAINS, POSITIFS ET STRICTEMENT
INFERIEURS A 1000. JE NE VERIFIERAI PAS CES CONDITIONS ET, DES
LORS, EN CAS D'ERREUR DE VOTRE PART, JE RISQUE DE ME PLANTER
OU DE DONNER DES REPONSES INCORRECTES.

Appuyez Entrée pour poursuivre
```

```
Donnez le premier nombre en chiffres romains :
IX
Donnez le deuxième nombre en chiffres romains :
VII

Voici la somme de ces deux nombres : XVI

Et leur différence : II

Et leur produit : LXIII

   On reprend avec un autre calcul ? (O ou N)
```

```
Donnez le premier nombre en chiffres romains :
CMXIX
Donnez le deuxième nombre en chiffres romains :
V

Voici la somme de ces deux nombres : CMXXIV

Et leur différence : CMXIV

Et leur produit : I°V°DXCV

   On reprend avec un autre calcul ? (O ou N)
```

 © CeFIS - FUNDP 2002

```
Donnez le premier nombre en chiffres romains :
XXI
Donnez le deuxième nombre en chiffres romains :
CXI

Voici la somme de ces deux nombres : CXXXII

Et leur différence : — XC

Et leur produit : MMCCCXXXI

   On reprend avec un autre calcul ? (O ou N)
```

```
Donnez le premier nombre en chiffres romains :
XLVII
Donnez le deuxième nombre en chiffres romains :
XLVII

Voici la somme de ces deux nombres : XCIV

La différence est nulle. Je ne peux l'écrire en chiffres romains

Et leur produit : MMCCIX

   On reprend avec un autre calcul ? (O ou N)
```

Le premier volume d'Images pour programmer avait insisté sur l'étape des spécifications (= du Quoi faire ?). Dorénavant, et sans entrer dans une formalisation excessive de cette étape (même si c'est cela qu'apprennent surtout à faire les informaticiens au cours de leur formation), nous nous laisserons guider par une triple préoccupation :

- Que doivent être les **entrées** (= les interventions de l'utilisateur, dans une perspective interactive comme celle-ci) ? ; quels sont les types d'entrées attendus, les limites fixées aux valeurs qu'elles peuvent prendre,... ? ; quelles vérifications de la conformité des données le programme à réaliser doit il envisager ? ;...
- Quelles sont les **sorties** attendues : formes et contenus des écrans affichés ?
- Quelles sont les **traitements** à envisager, en d'autres termes quelles sont les règles (souvent d'abord floues, implicites ou empiriques et qu'il faudra donc préciser) qui permettent de passer des entrées (données) aux sorties (résultats) ?

Avec l'apparition récente dans l'univers informatique du concept d'utilisateur naïf et le développement des interfaces graphiques qui facilitent le dialogue de celui-ci avec le système, les préoccupations de programmation se sont déplacées des traitements à commander vers la gestion des entrées-sorties. Il y a quelques années, 90% d'un programme s'attachait à l'analyse des traitements et 10% à celle du dialogue entre l'application et l'utilisateur; la proportion s'est très largement rééquilibrée aujourd'hui : on est devenu extrêmement attentif aux modalités des échanges entre l'utilisateur et l'exécutant. Dans une perspective interactive, programmer c'est avant tout aujourd'hui gérer un dialogue.

De plus, les questions émanent de deux champs différents : le monde de la tâche et le monde de l'exécutant.

- Les questions émanant de l'univers de la **tâche** sont celles que n'importe qui poserait s'il était confronté à la tâche dont il est question, sans aucune référence au fait que cette tâche devra être programmée. Ce sont par exemple ici les précisions qu'on pourrait solliciter à propos du travail consistant à manipuler des nombres en chiffres romains.

- Les question émanant de l'univers de **l'exécutant** sont celles qui ne sont posées que parce qu'on introduit, dans l'accomplissement de cette tâche, des contraintes relatives à l'exécutant-ordinateur. Il faut avoir intégré, par la pratique de la programmation, les caractéristiques propres aux traitements permis par l'ordinateur pour poser ici les bonnes questions.

En d'autres termes, la description du "Quoi faire ?" se fera dorénavant en complètant un tableau :

	Monde de la tâche	*Monde de l'exécutant*
Entrées	Questions de type 1	Questions de type 2
Traitements	Questions de type 3	Questions de type 4
Sorties	Questions de type 5	Questions de type 6

Par exemple, pour la tâche de manipulation des nombres en chiffres romains :

1. Entrées - Monde de la tâche :

   De quelle grandeur pourront être les nombres "romains" fournis ? : compris entre I (1) inclus et CMXCIX (999) inclus.

   Quels sont les symboles qui permettent l'écriture des nombres ? : ce sont I (i majuscule), pour 1, V pour 5, X pour 10, L pour 50, C pour 100, D pour 500 et M pour 1000.

2. Entrées - Monde de l'exécutant :

   D'autres caractères que les chiffres romains (comme l'espace) peuvent-ils figurer dans l'écriture des nombres ? : non, les nombres devront impérativement comporter les seuls symboles définis ci-dessus.

   Les entrées doivent-elles être validées : vérification de taille, de correction des nombres (conformité aux règles d'écriture des nombres "romains"[12]),...? : aucune validation n'est attendue; tant pis si le programme "se plante" à cause d'une donnée non correcte fournie par l'utilisateur.

   Du point de vue du programmeur, c'est évidemment une fameuse facilité que de ne pas devoir analyser la tâche consistant à vérifier qu'une succession de caractères constitue bien un nombre en chiffres romains correctement formé. C'est cependant un problème extrêmement intéressant et que je vous recommande d'envisager, soit à l'issue du présent problème, soit dans le cours d'un chapitre à venir consacré au traitement des chaînes de caractères.

3. Traitements - Monde de la tâche

   Quelles sont les règles d'écriture des nombres en chiffres romains ?

   Répondre de manière précise et synthétique à cette question, c'est évidemment écrire une partie essentielle du travail de programmation qui doit être fait. La réponse sera donc non pas "voici les règles d'écriture des nombres romains", mais plutôt "faites (faire !) comme je vous le montre sur quelques exemples". Cette traque des règles formelles sur base d'un fatras d'exemples est au coeur du processus de programmation et, plus largement, de l'informatique toute entière. Dire le "Quoi" en expliquant le "Comment", c'est faire la plus grosse partie du travail...qui doit être effectué ici.

   Les règles transparaissent dans les exemples suivants :

   - Comme précisé ci dessus, les signes utilisés sont I (i majuscule), pour 1, V pour 5, X pour 10, L pour 50, C pour 100, D pour 500 et M pour 1000;

---

12 Pour des raisons de concision, je parlerai dorénavant de "nombres romains" plutôt que de "nombres écrits en chiffres romains".

- On utilise d'abord le symbole I (1) pour I (1), II (2) et III (3);
- ensuite on écrit IV soit I (1) retiré de V (5), soit 4;
- puis V (5), VI (6), VII (7), VIII (8);
- puis IX soit I (1) retiré de X (10), soit 9;
- puis X (10), XI (11), XII (12), XIII (13), XIV (14), XV (15),...
- puis XIX (19), XX (20), XXI (21), ...XXXIX (39);
- et XL soit X (10) retiré de L (50), soit 40;
- puis XLI (41) ... L (50), ...LX (60), LXI (61),... LXIX (69), ..., LXXX (80), LXXXI (81),... LXXXIX (89);
- puis XC soit X (10) retiré de C (100), soit 90;
- XCIX (99), C (100), ...
- CC (200), CCI (201),...CCC (300), ... CCCXL (340), CCCXLI (341),... CCCXCIX (399);
- CD (400), ... CDXCIX (499), D (500), ...
- CM (900), ...CMXCIX (999), M (1000) et ainsi de suite jusque MMMCMXCIX (3999).

Puis, au delà on passe à la représentation en 2 parties, comme décrit ci-dessous.

Que faire lorsque la différence est 0 ? : Comme les romains ne connaissaient pas le 0, on fera afficher un message signalant que la réponse ne peut être donnée.

Et lorsque la différence est négative ? On affichera la valeur absolue précédée du signe —.

4. Traitements - Monde de l'exécutant

Le seul fait à signaler est la reprise du traitement à la demande de l'utilisateur.

5. Sorties - Monde de la tâche

Les règles d'écriture des nombres romains jusque MMMCMXCIX (3999) sont classiques. Comme il n'y a pas de symbole pour désigner 5000 (et au delà), comment écrire les produits qui seront obtenus et qui dépasseront 4000 ? : Le nombre à écrire doit alors être présenté en 2 parties successives : la première contient la tranche multipliant 1000, la seconde la tranche inférieure à 1000. Les symboles constituant la première tranche doivent être surlignés. Ainsi, on écrira :

$\overline{\mathrm{XXXIV}}\mathrm{CMXLVI}$ pour 34946

ou

$\overline{\mathrm{DCCXXIII}}\mathrm{CCCVI}$ pour 723306

La notation utilisée a varié au cours du temps et n'est pas uniforme (surtout pour les grands nombres). On consultera par exemple à ce propos le chapitre 16 de [Ifrah 94].

6. Sorties - Monde de l'exécutant

Peut-on utiliser une autre notation que le surlignement pour la partie du nombre multipliant M (1000) dans le cas de nombres supérieurs ou égaux à 4000 ? : Oui, chaque symbole de la partie du nombre multipliant 1000 sera accompagné du symbole °.

Ainsi, $\overline{\mathrm{XXXIV}}\mathrm{CMXLVI}$ sera noté X°X°X°I°V°CMXLVI et

$\overline{\mathrm{DCCXXIII}}\mathrm{CCCVI}$ s'écrira D°C°C°X°X°I°I°I°CCCVI

?

> 1. Pourquoi cette question est-elle posée ?
> 2. Avec cette convention quelle est la longueur (en nombre de caractères) du plus long nombre romain affiché et quel est-il ?

Les écrans présentés page 118 montrent clairement qu'un premier affichage doit avertir l'utilisateur de l'objectif de ce programme; les écrans suivants illustrent la forme attendue pour les sorties.

Nous savons à présent ce qui est attendu; il ne reste plus, à ce premier niveau d'analyse, qu'à trouver "Comment faire ?".

1.3 La calculatrice : Comment faire (à un premier niveau) ?

A ce tout premier niveau,

- On prend connaissance du premier nombre en chiffres romains à traiter
- On prend connaissance du second nombre en chiffres romains à traiter
- On décrypte le premier nombre pour évaluer le nombre en notation décimale correspondant
- On décrypte à son tour le second nombre pour évaluer le nombre en notation décimale qui lui correspond
- on additionne, soustrait et multiplie les 2 nombres décimaux obtenus
- on transforme en chiffres romains la somme décimale
- on transforme en chiffres romains la valeur absolue de la différence décimale
- on fait de même pour le produit décimal
- on fournit, en chiffres romains, la somme, le produit et la différence obtenue (en étant attentif au 0 et au signe — éventuel)

Le tout est à reprendre jusqu'à ce que l'utilisateur signale que c'est fini.

On objectera sans doute qu'on n'est guère avancé et que les deux opérations les plus compliquées : transformer un nombre "romain" en nombre "décimal" et l'inverse restent à analyser.

Et pourtant, la phrase même qui vient d'être écrite nous place au coeur de ce que j'aimerais vous faire découvrir ici : on ne parle plus de "transformer le premier nombre romain" ou de "transformer le second", mais de "transformer ***un*** nombre romain". De même, la transformation de la somme, du produit et de la différence a fait place à "transformer ***un*** nombre décimal en nombre romain".

1.4 La calculatrice : Comment faire faire (à un premier niveau) ?

Il nous reste à mettre en oeuvre dans un choix de variables et dans l'écriture d'une marche à suivre la stratégie retenue.

1.4.1 Structure de données.

1.4.1.1 Constantes inhérentes au problème.

A ce niveau de l'analyse, la seule constante qu'on pourrait éventuellement mettre au jour est la taille maximale des chaînes de caractères qui constitueront les nombres romains. Ce n'est certes pas une constante importante pour la compréhension du problème.

1.4.1.2 Types

Comme Pascal nous le permettra, nous définirons le type NombreRomain comme synonyme de string[40]. Autrement dit, les nombres romains manipulés seront toujours des chaînes d'au plus 40 caractères.

> On verra dans la suite que cette définition d'un type, qui paraît assez facultative, s'avérera indispensable, pour des raisons qui tiennent à la syntaxe exigée par Pascal.

1.4.1.3 Les tableaux nécessaires

Il n'y en a aucun à ce niveau de l'analyse.

> A nouveau, il ne faut pas anticiper sur les actions (complexes !) qui resteront à analyser : où nous en sommes, aucun tableau n'est indispensable.

1.4.2 Marche à suivre

```
AVERTIR
Répéter
    Lis PremierNombreRomain                                PremierNombreRomain de type NombreRomain
    Lis SecondNombreRomain                                 SecondNombreRomain de type NombreRomain
    DECIMALISER_PREMIERNOMBREROMAIN_EN_PREMIERNOMBREDECIMAL
                                                           PremierNombreDecimal de type entier long
    DECIMALISER_SECONDNOMBREROMAIN_EN_SECONDNOMBREDECIMAL
                                                           SecondNombreDecimal de type entier long
    SommeDecimal ← PremierNombreDecimal + SecondNombreDecimal          SommeDecimal entier long
    DifferenceDecimal ← PremierNombreDecimal — SecondNombreDecimal     DifferenceDecimal entier long
    ProduitDecimal ← PremierNombreDecimal * SecondNombreDecimal        ProduitDecimal entier long
    ROMANISER_SOMMEDECIMAL_EN_ SOMMEROMAIN
                                                           SommeRomain de type NombreRomain
    ROMANISER_ABS_DIFFERENCEDECIMAL_EN_ DIFFERENCEROMAIN
                                                           DifferenceRomain de type NombreRomain
    ROMANISER_PRODUITDECIMAL_EN_ PRODUITROMAIN
                                                           ProduitRomain de type NombreRomain
    Affiche SommeRomain
    Si DifferenceDecimal = 0 alors
        Affiche "Différence nulle"
    sinon
        Si DifferenceDecimal > 0 alors
            Affiche DifferenceRomain
        sinon
            Affiche "-" suivi de DifferenceRomain
    Affiche ProduitRomain
    Demander à l'utilisateur s'il souhaite recommencer et lire Reponse
jusqu'à ce que Reponse= 'N'                                Reponse de type char
```

> Les équivalents décimaux des nombres romains traités ont tous le type *entier long*. Nous savons que le type entier simple (integer) ne couvre que les nombres entiers dans l'intervalle -32768 à 32767 et que, si cela est suffisant en ce qui concerne les nombres fournis (compris entre 1 et 999) comme en ce qui concerne la somme et la différence de ceux-ci, leur produit par contre posera des problèmes puisqu'il peut atteindre 998001(soit 999 x 999). Pascal offre pour la manipulation des entiers de grande taille le type "entier long" (LongInt) qui permet la représentation d'entiers entre -2147483648 et 2147483647 (voir page 29). Même si ce n'est indispensable que dans le cas du produit, j'ai préféré, pour des motifs de simplicité et d'homogénéité, déclarer tous les entiers traités de type entier long.

En laissant de côté, les spécifications de AVERTIR (pour lesquelles il suffit de consulter le premier écran proposé), il reste à préciser :

DECIMALISER_PREMIERNOMBREROMAIN_EN_PREMIERNOMBREDECIMAL

PremierNombreRomain de type NombreRomain →	Transformer le nombre en chiffres romains *PremierNombreRomain* en son équivalent décimal et placer celui-ci dans *PremierNombreDecimal*	→ *Premier NombreDecimal* de type entier long

avec

Avant	Après
- *PremierNombreRomain* de type NombreRomain qui contient un nombre romain correctement écrit (entre I (1) et CMXCIX (999)).	- *PremierNombreRomain* ne peut avoir changé - *PremierNombreDecimal* de type entier long, contient un entier entre 1 et 999 correspondant au nombre *PremierNombreRomain* .

et, bien entendu :

DECIMALISER_SECONDNOMBREROMAIN_EN_SECONDNOMBREDECIMAL

SecondNombreRomain de type NombreRomain →	Transformer le nombre en chiffres romains *SecondNombreRomain* en son équivalent décimal et placer ce dernier dans *SecondNombreDecimal*	→ *SecondNombreDecimal* de type entier long

avec

Avant	Après
- *SecondNombreRomain* de type NombreRomain qui contient un nombre romain correctement écrit (entre I (1) et CMXCIX (999)).	- *SecondNombreRomain* ne peut avoir changé - *SecondNombreDecimal* de type entier long, contient un entier entre 1 et 999 correspondant au nombre *SecondNombreRomain* .

et

ROMANISER_SOMMEDECIMAL_EN_ SOMMEROMAIN

SommeDecimal → de type entier long	Transformer le résultat *SommeDecimal* en son équivalent en chiffres romains *SommeRomain*	→ *SommeRomain* de type NombreRomain

avec

Avant	Après
- *SommeDecimal* de type entier long, qui contient la somme écrite sous forme décimale, toujours comprise entre 2 et 1998.	- *SommeDecimal* ne peut avoir changé - *SommeRomain* de type NombreRomain, contenant le nombre en chiffres romains correspondant à *SommeDecimal*.

ROMANISER_ABS_DIFFERENCEDECIMAL_EN_ DIFFERENCEROMAIN

DifferenceDecimal → de type entier long	Transformer la valeur absolue du résultat *DifferenceDecimal* en son équivalent en chiffres romains *DifferenceRomain*	→ *DifferenceRomain* de type NombreRomain

avec

Avant	Après
- *DifferenceDecimal* de type entier long, qui contient la différence écrite sous forme décimale, toujours comprise entre -998 et 998.	- *DifferenceDecimal* ne peut avoir changé - *DifferenceRomain* de type NombreRomain, contenant le nombre en chiffres romains correspondant à la valeur absolue de *DifferenceDecimal*.

ROMANISER_PRODUITDECIMAL_EN_ PRODUITROMAIN

ProduitDecimal → de type entier long	Transformer le résultat *ProduitDecimal* en son équivalent en chiffres romains *ProduitRomain*	→ *ProduitRomain* de type NombreRomain

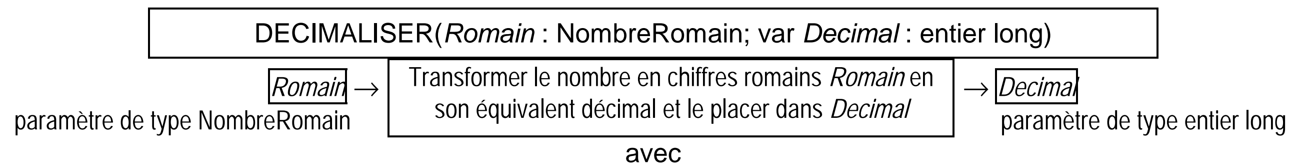
avec

Avant	Après
- *ProduitDecimal* de type entier long, qui contient le produit écrit sous forme décimale, toujours compris entre 1 et 998001 (999*999)	- *ProduitDecimal* ne peut avoir changé - *ProduitRomain* de type NombreRomain, contenant le nombre en chiffres romains correspondant à *ProduitDecimal*.

On pourrait poursuivre l'analyse comme si de rien n'était avec chacune des actions complexes ainsi spécifiée. On constate cependant que les deux premières actions complexes DECIMALISER_PREMIERNOMBREROMAIN_EN_PREMIERNOMBREDECIMAL et DECIMALISER_SECONDNOMBREROMAIN_EN_SECONDNOMBREDECIMAL constituent la *même* action, appliquée la première fois à *PremierNombreRomain* pour remplir *PremierNombre Decimal*, la seconde fois à *SecondNombreRomain* pour remplir *SecondNombreDecimal*. Il serait donc absurde de recommencer deux fois la même analyse, les seules modifications de l'une à l'autre étant les changements du nom des variables concernées.

Ce qu'il faudrait, c'est non pas pouvoir analyser l'action permettant, sur base de telle variable **précise** contenant un nombre en chiffres romains de passer à telle variable **précise** contenant son équivalent décimal, mais décrire l'action qui sur base d'*un* nombre en chiffres romains fournisse dans **une quelconque** variable (de type entier long) son équivalent décimal. C'est seulement aux moments où cette action serait activée (= lors de l'appel de la procédure) qu'on préciserait à chaque fois **le** nombre à transformer et **la** variable accueillant le résultat de la transformation.

On écrirait en quelque sorte :



avec

Avant	Après
- *Romain* : paramètre de type NombreRomain qui contient un nombre romain correctement écrit (entre I (1) et CMXCIX (999)).	- *Decimal* : paramètre de type entier long, contient un entier entre 1 et 999 correspondant au nombre en chiffres romains *Romain* .

Au moment de l'appel des procédures, au lieu d'appeler successivement deux procédures différentes, c'est la *même* procédure, DECIMALISER, qui serait appelée, mais en précisant à chaque fois ce que sont vraiment chacun des deux paramètres dont elle est assortie.

Ainsi, au lieu d'appeler

DECIMALISER_PREMIERNOMBREROMAIN_EN_PREMIERNOMBREDECIMAL, on appellerait DECIMALISER, en précisant, pour cette fois que

- *PremierNombreRomain* "remplace" *Romain* et
- *PremierNombreDecimal* "remplace" *Decimal*

et au lieu de

DECIMALISER_SECONDNOMBREROMAIN_EN_SECONDNOMBREDECIMAL, on appellerait à nouveau DECIMALISER, en précisant, à ce second appel, que

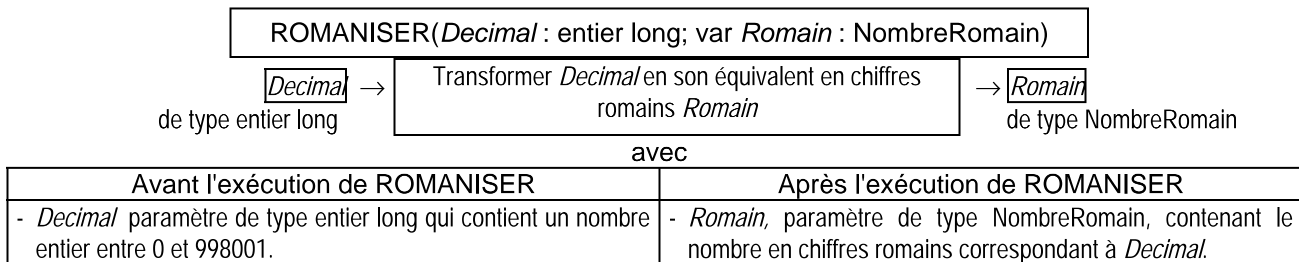
- *SecondNombreRomain* "remplace" *Romain* et
- *SecondNombreDecimal* "remplace" *Decimal.*

Romain et *Decimal* sont ce que nous appellerons des paramètres de la procédure DECIMALISER. Ils sont encadrés dans le schéma proposé pour insister sur le fait qu'**ils font partie de la procédure** spécifiée. A chaque appel de DECIMALISER, ces deux paramètres se verront associer des "valeurs" précises. Nous allons bien entendu revenir longuement sur ce concept

nouveau et important, mais il est indispensable d'avoir compris dès à présent à quoi ces paramètres sont utiles.

> Il est inutile, pour l'instant, de se préoccuper des détails syntaxiques. Ainsi, le fait que le paramètre *Decimal* est précédé de la mention var ne sera expliqué que plus loin (voir page 159).

De la même manière, les trois dernières actions complexes peuvent se résumer en une seule :



Avant l'exécution de ROMANISER	Après l'exécution de ROMANISER
- *Decimal* paramètre de type entier long qui contient un nombre entier entre 0 et 998001.	- *Romain,* paramètre de type NombreRomain, contenant le nombre en chiffres romains correspondant à *Decimal.*

> Il faut noter que ces deux paramètres *Decimal* et *Romain* accompagnant ROMANISER n'ont rien à voir (même s'ils portent des noms identiques) avec les deux paramètres *Romain* et *Decimal* accompagnant DECIMALISER.

Et au moment des appels, on appellerait à trois reprises la même procédure ROMANISER, mais on préciserait :

- pour la transformation de la somme :
  - *Decimal* est *SommeDecimal* et
  - *Romain* est à comprendre comme *SommeRomain;*
- en ce qui concerne la différence :
  - *Decimal* est la valeur absolue de *DifferenceDecimal;*
  - *Romain* est à comprendre comme *DifferenceRomain;*

    Il faut bien noter que ROMANISER ayant comme tâche de transformer le nombre décimal qu'on lui passe (à travers le paramètre *Decimal*), il faudra, lors de l'appel destiné à "romaniser" la différence contenue dans *DifferenceDecimal*, remplacer le paramètre *Decimal* non par *DifferenceDecimal* mais par la valeur absolue de cette dernière.
- en ce qui concerne le produit :
  - *Decimal* est *ProduitDecimal* et
  - *Romain* est en réalité *ProduitRomain.*

Avec cette réécriture qui fait seulement apparaître deux procédures ROMANISER et DECIMALISER, assorties de paramètres, là où 5 procédures avaient primitivement été isolées, la marche à suivre principale devient :

AVERTIR
Répéter
    Lis *PremierNombreRomain* — *PremierNombreRomain,*
    Lis *SecondNombreRomain* — *SecondNombreRomain* de type NombreRomain
    DECIMALISER(*PremierNombreRomain*, *PremierNombreDecimal*)
        PremierNombreDecimal de type entier long
    DECIMALISER(*SecondNombreRomain*, *SecondNombreDecimal*)
        SecondNombreDecimal de type entier long
    SommeDecimal ← *PremierNombreDecimal* + *SecondNombreDecimal* — *SommeDecimal* de type entier long
    DifferenceDecimal ← *PremierNombreDecimal* -*SecondNombreDecimal* — *DifferenceDecimal* entier long

 © CeFIS - FUNDP 2002

ProduitDecimal ← *PremierNombreDecimal* * *SecondNombreDecimal* — *ProduitDecimal* entier long

ROMANISER(*SommeDecimal*, *SommeRomain*) — *SommeRomain* de type NombreRomain

ROMANISER(abs(*DifferenceDecimal*), *DifferenceRomain*) — *DifferenceRomain* de type NombreRomain

ROMANISER(*ProduitDecimal*, *ProduitRomain*) — *ProduitRomain* de type NombreRomain

Affiche *SommeRomain*
Si *DifferenceDecimal* = 0 alors
  Affiche "Différence nulle"
sinon
  Si *DifferenceDecimal* > 0 alors
    Affiche *DifferenceRomain*
  sinon
    Affiche "-" suivi de *DifferenceRomain*
Affiche *ProduitRomain*
Demander à l'utilisateur s'il souhaite recommencer et lire *Reponse*
jusqu'à ce que *Reponse*= 'N' — *Reponse* de type caractère

On notera dès à présent que lors de l'appel des procédures assorties de paramètres, chacun des paramètres définis dans les spécifications (= la description du "Quoi faire ?") de ces procédures se voit associer une information ou une variable du programme principal. Il faut donc mettre en parallèle la **définition** de la procédure et de ses paramètres et son **appel**, qui précise les valeurs prises, lors de cet appel, par les paramètres. Ainsi, par exemple :

Définition : `DECIMALISER( Romain Decimal )`

1er appel : `DECIMALISER(PremierNombreRomain,PremierNombreDecimal)`

Définition : `DECIMALISER( Romain Decimal )`

2ème appel : `DECIMALISER(SecondNombreRomain,SecondNombreDecimal)`

1.5 La calculatrice : Comment dire (à un premier niveau) ?

Et voici, à ce stade de l'analyse, le texte du programme principal correspondant. La procédure AVERTIR, dont le contenu est suffisamment immédiat, y a déjà été adjointe.

```
program CALCULATRICE_POUR_JULES;
        (* Ce programme fait lire deux nombres en chiffres romains (positifs et inférieurs à 1000).
        Il fournit ensuite, toujours en chiffres romains, leur somme, leur différence et leur
        produit. Il y a passage par une écriture habituelle des nombres et transformations, dans un
        sens puis dans l'autre. *)

uses WinCrt;

type NombreRomain=string[40];
        (* chaînes de caractères représentant un nombre en chiffres romains *)

var PremierNombreRomain,
     SecondNombreRomain,
     SommeRomain,
     DifferenceRomain,
     ProduitRomain : NombreRomain;
        (* il s'agit des divers nombres, en chiffres romains, qui seront manipulés *)
     PremierNombreDecimal,
     SecondNombreDecimal,
```

```
    SommeDecimal,
    DifferenceDecimal,
    ProduitDecimal : Longint;
       (* Il s'agit des correspondants décimaux des précédents *)
    Reponse:char;

procedure AVERTIR;
begin
Clrscr;
writeln('Vous allez me fournir 2 nombres en chiffres romains et je vous');
writeln('fournirai, toujours en chiffres romains, leur somme, leur différence');
writeln('et leur produit.');
writeln('ATTENTION, LES DEUX NOMBRES QUE VOUS ME DONNEZ DOIVENT ETRE ');
writeln('CORRECTEMENT ECRITS EN CHIFFRES ROMAINS, POSITIFS ET STRICTEMENT');
writeln('INFERIEURS A 1000. JE NE VERIFIERAI PAS CES CONDITIONS ET, DES');
writeln('LORS, EN CAS D''ERREUR DE VOTRE PART, JE RISQUE DE ME PLANTER');
writeln('OU DE DONNER DES REPONSES INCORRECTES.');
writeln;writeln('Appuyez Entrée pour poursuivre');
readln;
end;

procedure DECIMALISER(Romain : NombreRomain; var Decimal : LongInt);          (1)
       (* Elle fait transformer le contenu du paramètre valeur Romain (de type NombreRomain) en son
       équivalent décimal Decimal (paramètre variable) *)

procedure ROMANISER(Decimal: LongInt; var Romain:NombreRomain);               (1)
       (* Elle fait transformer le nombre entier long Decimal (paramètre valeur) en une chaîne
       contenant ce nombre en chiffres romains : Romain (de type NombreRomain) *)

begin (* début du programme principal *)
AVERTIR;
repeat
  Clrscr;
  writeln('Donnez le premier nombre en chiffres romains : ');
  readln(PremierNombreRomain);
  writeln('Donnez le deuxième nombre en chiffres romains :');
  readln(SecondNombreRomain);
  DECIMALISER(PremierNombreRomain,PremierNombreDecimal);                      (2)
  DECIMALISER(SecondNombreRomain,SecondNombreDecimal);                        (2)
  SommeDecimal:=PremierNombreDecimal+SecondNombreDecimal;
  DifferenceDecimal:=PremierNombreDecimal-SecondNombreDecimal;
  ProduitDecimal:=PremierNombreDecimal*SecondNombreDecimal;
  ROMANISER(SommeDecimal,SommeRomain);                                        (2)
  ROMANISER(abs(DifferenceDecimal),DifferenceRomain);                         (2)
  ROMANISER(ProduitDecimal,ProduitRomain);                                    (2)
  writeln;
  writeln('Voici la somme de ces deux nombres : ', SommeRomain);
  writeln;
  if DifferenceDecimal=0 then
      writeln('La différence est nulle. Je ne peux l''écrire en chiffres romains')
  else
     write('Et leur différence : ');
     if DifferenceDecimal<0 then
        writeln('-',DifferenceRomain)
     else
        writeln(DifferenceRomain);
  writeln;
  writeln('Et leur produit : ',ProduitRomain);
  Gotoxy(4,20);
  writeln('On reprend avec un autre calcul ? (O ou N)');
  readln(Reponse);
  Reponse:=upcase(Reponse);
until Reponse='N';
end.
```

1.5.1 Commentaires sur le texte du programme

(1) On trouve bien, à la suite de l'identification de chaque procédure, la liste des paramètres qui y sont attachés. On notera les points-virgules séparant ces derniers. La signification du mot var qui précède certains paramètres sera expliquée plus tard.

(2) On constate que lors de l'appel, la liste des variables qui doivent être associées aux paramètres est fournie entre parenthèses. Les types doivent bien entendu se correspondre et l'ordre des valeurs ou des variables associées suit simplement celui des paramètres qu'elles "remplacent". On notera encore les virgules qui séparent les éléments de la liste.

1.6 Analyse de DECIMALISER

1.6.1 DECIMALISER ? Quoi faire ?

Les spécifications sont décrites à la page 125. Essentiellement, il s'agit de transformer un chiffre romain présent dans *Romain* (de type NombreRomain) en son équivalent décimal *Decimal* (de type entier long).

1.6.2 DECIMALISER ? Comment faire ?

Face à un nombre en chiffres romains, comme par exemple MMCDLXXXVII, la stratégie à mettre en oeuvre est finalement assez immédiate.

Il faut en tout cas garder (dans un coin de la tête ou griffonné sur un bout de papier) les valeurs correspondant aux divers symboles intervenant dans l'écriture en chiffres romains.

On retient donc la correspondance :

I	V	X	L	C	D	M
1	5	10	50	100	500	1000

On va alors parcourir le nombre romain **de droite à gauche** : à chaque fois qu'on trouve un symbole, on en comptabilise la valeur (après un coup d'oeil au tableau ci-dessus) en ajoutant la valeur correspondant à ce symbole (ou en la retranchant).

Ceci postule bien entendu qu'il sera possible de faire parcourir par l'exécutant les divers caractères constituant les nombres en chiffres romains ; comme ces derniers sont du type NombreRomain, donc string[40], il faudra pouvoir extraire l'un après l'autre les caractères constituant une chaîne de caractères.

Ainsi, pour le nombre cité (MMCDLXXXVII), on trouve successivement :

I	Valeur : 1	Total décimal : 1
I	Valeur : 1	Total décimal : 2
V	Valeur : 5	Total décimal : 7
X	Valeur : 10	Total décimal : 17
X	Valeur : 10	Total décimal : 27
X	Valeur : 10	Total décimal : 37
L	Valeur : 50	Total décimal : 87
D	Valeur : 500	Total décimal : 587

puis on trouve C (100) **qui doit être retranché** et non ajouté, puisqu'il précède D (500) dans l'écriture du nombre en chiffres romains (= dans le parcours de droite à gauche, on l'a rencontré après D **et sa valeur est inférieure** à D). Ainsi :

C Valeur : 100 Total décimal : 487, car 100 est à retrancher

M Valeur : 1000 Total décimal : 1487
M Valeur : 1000 Total décimal : 2487

Ainsi, la règle à suivre est immédiate :

- On parcourt le nombre de droite à gauche, symbole après symbole
- A chaque symbole rencontré :
  - on évalue la valeur décimale correspondante;
  - si le symbole est "supérieur" au symbole précédemment rencontré (= sa valeur décimale est plus grande ou égale à celle du symbole précédent dans le parcours), on ajoute la valeur correspondante au décompte effectué;

    sinon (le symbole est "inférieur" au symbole rencontré juste avant), on retranche la valeur correspondante du décompte effectué.

1.6.3 DECIMALISER ? Comment faire faire ?

1.6.3.1 Choix des structures de données

Il nous faut essentiellement décider de la manière dont nous allons implémenter (= réaliser d'une manière qui sera exécutable) l'association entre les symboles romains et leurs valeurs, représentée dans le "tableau" décrit ci-dessus.

Faire "retenir" ces diverses valeurs par l'exécutant qui sera chargé de les manipuler nous incline évidemment à penser à la structure de tableau. Si les divers symboles (I, V, X,...) pouvaient servir d'étiquettes à des tiroirs, il suffirait d'y placer à chaque fois la valeur correspondante (1, 5, 10,...). Sur base du symbole, on repérerait aisément la valeur correspondante.

Ce qu'il nous faut donc décider, c'est comment nous rendons compte de ces divers symboles. Ce sont ceux qui permettent d'écrire les nombres en chiffres romains et nous serons tentés d'y voir, dès lors, des caractères. Ainsi I, V, X, L,... devraient être compris comme 'I', 'V', 'X', 'L',.... Malheureusement, cette suite de caractères ne peut servir en tant que telle à étiqueter les tiroirs d'un tableau.

> Rappelons à nouveau que le type d'information pouvant servir à indicer (= étiqueter) un tableau doit être tel que la fonction succ (successeur) ait du sens : il doit être de type **scalaire**. De plus, pour préciser ces indices, on se contente (en Pascal) de donner le premier et le dernier; toutes les valeurs successivement rencontrées entre ce premier et ce dernier constituent les divers indices tu tableau.

En effet, si le type caractère (char) peut servir à indicer un tableau, c'est à condition que nous en extrayions **tout un intervalle**, c'est à dire *tous* les caractères compris entre un premier et un dernier. Et ce que nous voulons, ce n'est certes pas utiliser tous les caractères entre 'I' et 'M', ni non plus tous ceux de l'intervalle 'C'..'X'. Bref, les seuls caractères 'I', 'V', 'X', 'L', 'C', 'D' et 'M' ne peuvent en tant que tels servir à indicer un tableau.

Il nous reste bien entendu la possibilité, comme lors du problème du Poker précédemment abordé, de définir un **type énuméré**, propre à notre problème et qui serait constitué des constantes I, V, X, L, C, D et M, ces dernières n'ayant rien à voir avec les caractères 'I', 'V', 'X',....

Ayant donc défini un type **énuméré**

ChiffresRomains = (I, V, X, L, C, D, M),

nous sommes alors en mesure de créer un tableau :

Valeurs : array[ChiffresRomains] of integer, représenté par

			Valeurs			
1	5	10	50	100	500	1000
I	V	X	L	C	D	M

Il nous faut cependant, dès à présent, garder à l'esprit une difficulté : lorsque nous disposerons d'un nombre en chiffres romains, c'est d'une suite de caractères qu'il s'agira; il sera indispensable d'associer ces caractères ('I', 'V',...) aux constantes énumérées définies (I, V, ...) en nous disant bien que (contrairement à ce que nous pourrions croire), 'X' n'a, a priori, pour l'exécutant ordinateur, rien à voir avec X.

Comme les constantes d'un type énuméré ne peuvent être ni lues (au clavier), ni affichées, il est fréquent, lorsque une analyse conduit à la définition d'un type énuméré, de devoir prévoir des procédures de conversion : en entrée, pour passer des chaînes de caractères lues aux constantes énumérées correspondantes, en sortie, pour passer des constantes énumérées aux chaînes de caractères correspondantes.

1.6.3.2 Marche à suivre

Dans un premier temps, nous en écrirons une version qui garde un certain "flou" et qui ne soit pas optimale. En nous souvenant que nous disposons donc du tableau *Valeurs* et qu'il s'agit de décimaliser le contenu du paramètre *Romain* pour en placer l'équivalent décimal dans le paramètre *Decimal* :

REMPLIR_VALEURS

CaractereActuel ← *dernier caractère de Romain* — *CaractereActuel* de type caractère

TRANSFORMER_CARACTEREACTUEL_EN_INDICEACTUEL — *IndiceActuel* de type ChiffresRomains

Decimal ← *contenu du tiroir de Valeurs ayant comme étiquette IndiceActuel*

Pour *Compteur faisant parcourir Romain de l'avant dernier caractère jusqu'au premier caractère* — *Compteur* entier

  CaractereActuel ← *caractère n° Compteur de Romain*

  TRANSFORMER_CARACTEREACTUEL_EN_INDICEACTUEL

  CaractereSuivant ← *caractère n° Compteur + 1 de Romain* — *CaractereSuivant* de type caractère

  TRANSFORMER_CARACTERESUIVANT_EN_INDICESUIVANT — *Indicesuivant* de type ChiffresRomains

  Si *IndiceActuel* ≥ *Indicesuivant* alors **(1)**

    Decimal ← *Decimal* + la *composante d'indice IndiceActuel de Valeurs*

  sinon

    Decimal ← *Decimal* — *la composante d'indice IndiceActuel de Valeurs*

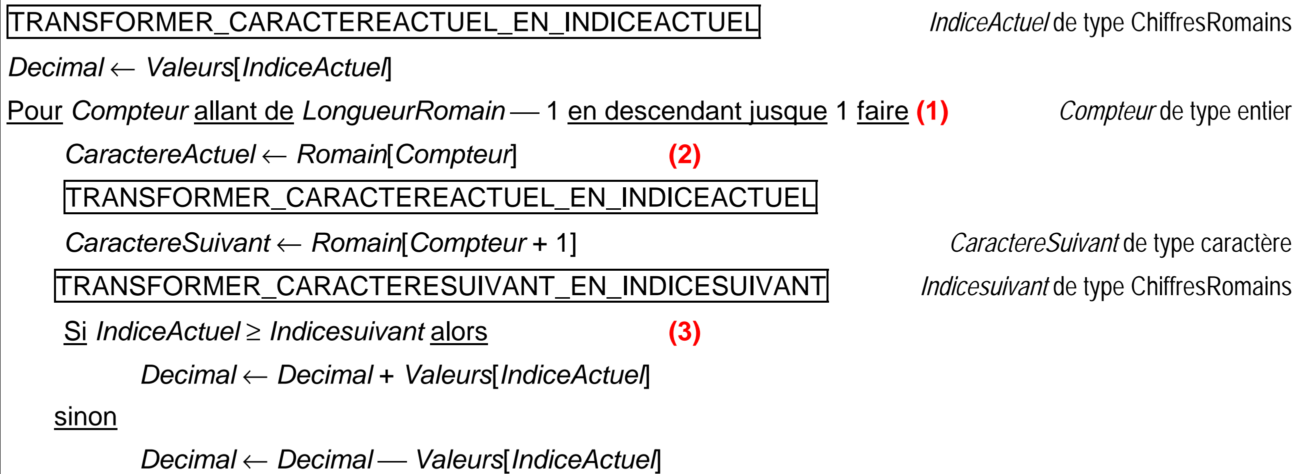
(1) Il faut se rappeler que l'ordre dans lequel les constantes d'un type énuméré sont écrites (ici I, V, X, L, C, D, M pour ChiffresRomains) définit de fait un **ordre** et qu'on peut donc comparer des indices de ce type (ou leur appliquer des fonctions comme succ et pred).

Dans la marche à suivre précédente, j'ai laissé sous une forme proche de notre mode d'expression certaines périphrases comme "Pour *Compteur faisant parcourir Romain de l'avant dernier caractère jusqu'au premier caractère*" ou "*la composante d'indice IndiceActuel de Valeurs*". Sous une forme moins floue, (en se souvenant que la fonction length fournit la longueur d'une chaîne de caractères, autrement dit le nombre de caractères qu'elle comporte), on peut réécrire :

REMPLIR_VALEURS

LongueurRomain ← length(*Romain*) — *LongueurRomain* de type entier

CaractereActuel ← *Romain*[*LongueurRomain*] **(4)** — *CaractereActuel* de type caractère



TRANSFORMER_CARACTEREACTUEL_EN_INDICEACTUEL — *IndiceActuel* de type ChiffresRomains

Decimal ← *Valeurs*[*IndiceActuel*]

Pour *Compteur* allant de *LongueurRomain* — 1 en descendant jusque 1 faire **(1)** — *Compteur* de type entier

    CaractereActuel ← *Romain*[*Compteur*] **(2)**

    TRANSFORMER_CARACTEREACTUEL_EN_INDICEACTUEL

    CaractereSuivant ← *Romain*[*Compteur* + 1] — *CaractereSuivant* de type caractère

    TRANSFORMER_CARACTERESUIVANT_EN_INDICESUIVANT — *Indicesuivant* de type ChiffresRomains

    Si *IndiceActuel* ≥ *Indicesuivant* alors **(3)**

        Decimal ← *Decimal* + *Valeurs*[*IndiceActuel*]

    sinon

        Decimal ← *Decimal* — *Valeurs*[*IndiceActuel*]

(1) Cette boucle *décroissante* Pour... ne sera pas effectuée si *LongueurRomain* - 1 (valeur initiale du *Compteur*) est strictement inférieure à 1 (valeur terminale du *Compteur*), donc si *LongueurRomain*-1 < 1 ou encore *LongueurRomain* ≤ 1, c'est à dire si la chaîne *Romain* contenant le nombre (et qui ne peut être vide) ne comporte qu'un seul caractère.

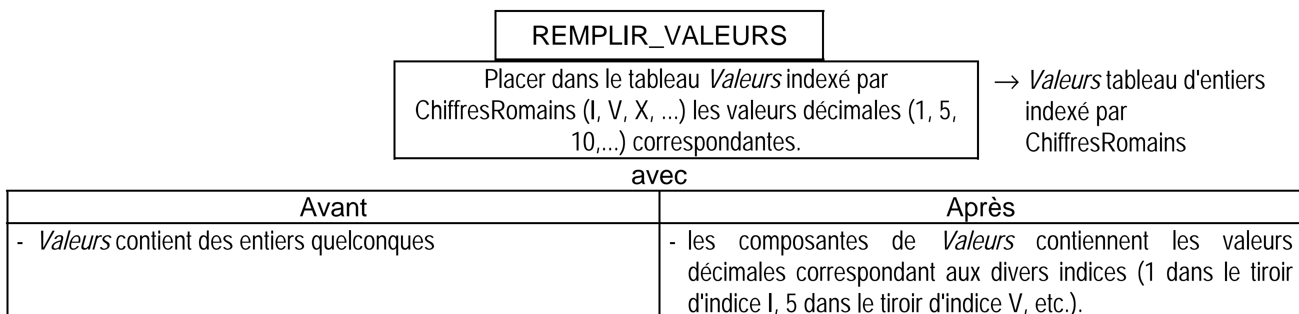
(2) On notera que pour accéder à un caractère d'une chaîne de caractères contenue dans une variable (ici *Romain*, de type NombreRomain, donc string[40]), on fait suivre le nom de cette variable du numéro du caractère souhaité. Ici *Romain*[*Compteur*] désigne donc le caractère numéro *Compteur* de la chaîne contenue dans *Romain*.

> Un chapitre (du tome 3!) sera consacré aux chaînes de caractères et aux outils permettant de les manipuler. Nous savons déjà que la fonction length, appliquée à une chaîne de caractères en fournit la longueur (= le nombre de caractères) et que pour désigner un caractère d'une chaîne, il suffit de placer entre crochets la position du caractère souhaité, à la suite de la chaîne envisagée. Ce dernier détail nous incline à penser que, derrière une variable de type chaîne de caractères (string), se cache probablement un tableau de caractères. (Voir l'annexe, page 207).

(3) Rappelons que *IndiceActuel* et *IndiceSuivant* sont tous deux du type énuméré ChiffresRomains. On peut donc comparer deux données de ce type, étant entendu que l'ordre sous-jacent est celui prescrit lors de la définition de ce type énuméré : I < V < X < ...

(4) Au tout début du processus, *CaractereActuel* contient le dernier caractère de la chaîne *Romain* (celui noté *Romain*[*LongueurRomain*]).

Comme d'habitude, il reste à spécifier les actions complexes qui sont apparues au sein de cette marche à suivre :



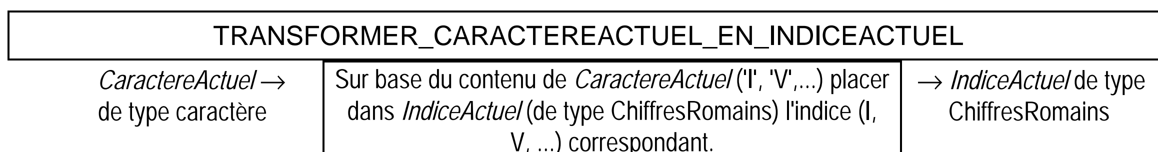
REMPLIR_VALEURS

Placer dans le tableau *Valeurs* indexé par ChiffresRomains (I, V, X, ...) les valeurs décimales (1, 5, 10,...) correspondantes. → *Valeurs* tableau d'entiers indexé par ChiffresRomains

avec

Avant	Après
- *Valeurs* contient des entiers quelconques	- les composantes de *Valeurs* contiennent les valeurs décimales correspondant aux divers indices (1 dans le tiroir d'indice I, 5 dans le tiroir d'indice V, etc.).

et



TRANSFORMER_CARACTEREACTUEL_EN_INDICEACTUEL

CaractereActuel → de type caractère

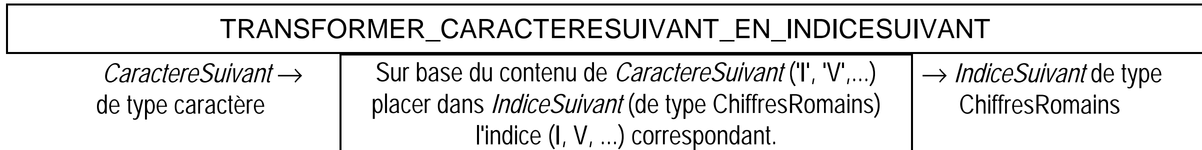
Sur base du contenu de *CaractereActuel* ('I', 'V',...) placer dans *IndiceActuel* (de type ChiffresRomains) l'indice (I, V, ...) correspondant.

→ *IndiceActuel* de type ChiffresRomains

avec

Avant	Après
- *CaractereActuel* contenant l'un des caractères utilisés dans l'écriture des nombres romains.	- *CaractereActuel* ne peut avoir changé - *IndiceActuel* de type ChiffresRomains, contient l'indice correspondant au caractère contenu dans *CaractereActuel.*

et enfin

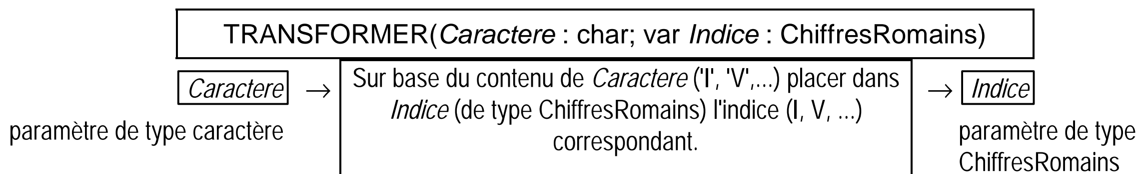


avec

Avant	Après
- *CaractereSuivant* contenant l'un des caractères utilisés dans l'écriture des nombres romains.	- *CaractereSuivant* ne peut avoir changé - *IndiceSuivant* de type ChiffresRomains, contient l'indice correspondant au caractère contenu dans *CaractereSuivant.*

Ces deux dernières actions me donnent, pour la seconde fois, l'occasion de souligner une similitude qui va conduire à unifier ces deux actions en une seule, assortie de paramètres qui vont permettre deux appels de cette action unique en l'appliquant chaque fois à des variables différentes.

On aura donc :



avec

Avant	Après
- *Caractere* paramètre contenant l'un des caractères utilisés dans l'écriture des nombres romains.	- *Indice* paramètre de type ChiffresRomains, contient l'indice correspondant au caractère contenu dans *Caractere*

Ce qui permet de réécrire la marche à suivre proposée sous la forme :

```
REMPLIR_VALEURS
LongueurRomain ← length(Romain)                          LongueurRomain de type entier
CaractereActuel ← Romain[LongueurRomain]                 CaractereActuel de type caractère
TRANSFORMER(CaractereActuel, IndiceActuel)               IndiceActuel de type ChiffresRomains
Decimal ← Valeurs[IndiceActuel]
Pour Compteur allant de LongueurRomain -1 en descendant jusque 1 faire    Compteur de type entier
    CaractereActuel ← Romain[Compteur]
    TRANSFORMER(CaractereActuel, IndiceActuel)
    CaractereSuivant ← Romain[Compteur + 1]              CaractereSuivant de type caractère
    TRANSFORMER(CaractereSuivant, IndiceSuivant)         IndiceSuivant de type ChiffresRomains
    Si IndiceActuel ≥ Indicesuivant alors
        Decimal ← Decimal + Valeurs[IndiceActuel]
    sinon
        Decimal ← Decimal - Valeurs[IndiceActuel]
```

On peut constater qu'au premier et au second appel de TRANSFORMER, c'est le contenu de *CaractereActuel* qui tient lieu de *Caractere* et c'est *IndiceActuel* qui est associé à *Indice*. Au

troisième appel, c'est le contenu de *CaractereSuivant* qui tient lieu de *Caractere* tandis que c'est *IndiceSuivant* qui est associé à *Indice*.

On aurait d'ailleurs pu analyser un peu différemment l'action de décimalisation (mais j'aurais alors perdu une occasion d'illustrer une seconde fois la nécessité de paramètres pour une procédure) :

REMPLIR_VALEURS

LongueurRomain ← length(*Romain*) — *LongueurRomain* de type entier

CaractereActuel ← *Romain*[*LongueurRomain*] — *CaractereActuel* de type caractère

TRANSFORMER_CARACTEREACTUEL_EN_INDICEACTUEL — *IndiceActuel* de type ChiffresRomains

Decimal ← *Valeurs*[*IndiceActuel*]

Pour *Compteur* allant de *LongueurRomain* -1 en descendant jusque 1 faire — *Compteur* de type entier

  IndiceSuivant ← *IndiceActuel* — *Indicesuivant* de type ChiffresRomains

  CaractereActuel ← *Romain*[*Compteur*]

  TRANSFORMER_CARACTEREACTUEL_EN_INDICEACTUEL

  Si *IndiceActuel* ≥ *Indicesuivant* alors

    Decimal ← *Decimal* + *Valeurs*[*IndiceActuel*]

  sinon

    Decimal ← *Decimal* - *Valeurs*[*IndiceActuel*]

En gardant la première analyse proposée, on peut dès à présent proposer :

1.6.4 DECIMALISER. Comment dire ?

```
procedure DECIMALISER(Romain : NombreRomain; var Decimal : LongInt);          (1)
       (* Elle fait transformer le contenu du paramètre valeur Romain (de type NombreRomain) en son
       équivalent décimal Decimal (paramètre variable) *)

type ChiffresRomains = (I,V,X,L,C,D,M);
       (* type énuméré reprenant les symboles intervenant dans l'écriture des nombres en chiffres
       romains *)

var Valeurs : array[ChiffresRomains] of integer;
       (* Tableau contenant les valeurs décimales des divers chiffres romains *)
    Compteur, (* pour le parcours de la chaîne  Romain *)
    LongueurRomain (* longueur de la chaîne Romain *)
       : integer;
    CaractereActuel,(* qui contiendra successivement chacun des caractères de la chaîne Romain *)
    CaractereSuivant (* qui, dans l'exploration de la chaîne Romain, contiendra le caractère
       suivant CaractereActuel *)
       : char;
    IndiceSuivant,(* qui contiendra la constante de type ChiffresRomains correspondant au
       caractère CaractereSuivant *)
    IndiceActuel (* qui contiendra la constante de type ChiffresRomains correspondant au caractère
       CaractereActuel *)
       : ChiffresRomains;

procedure REMPLIR_VALEURS;
       (* elle initialise le tableau Valeurs en y plaçant la valeur des divers chiffres romains : le
       tiroir I reçoit 1, V reçoit 5,...*)

procedure TRANSFORMER(Caractere: char; var Indice: ChiffresRomains);
       (* elle fait transformer le paramètre valeur Caractere de type char en l'indice Indice
       (paramètre de type ChiffresRomains) *)

begin  (* début de DECIMALISER *)
       (* initialisation du tableau Valeurs avec les valeurs des chiffres romains *)
REMPLIR_VALEURS;
LongueurRomain:=length(Romain);
CaractereActuel:=Romain[LongueurRomain];(* CaractereActuel contient le dernier symbole de
       Romain *)
```

```
TRANSFORMER(CaractereActuel,IndiceActuel);
Decimal:=Valeurs[IndiceActuel];
       (* on a placé dans Decimal l'équivalent du dernier chiffre (romain) de Romain *)
       (* on va explorer Romain de droite à gauche *)
for Compteur := LongueurRomain-1 downto 1 do
   begin
   CaractereActuel:=Romain[Compteur];
   TRANSFORMER(CaractereActuel,IndiceActuel);
   CaractereSuivant := Romain[Compteur+1];
   TRANSFORMER(CaractereSuivant,IndiceSuivant);
   if IndiceActuel>=IndiceSuivant then
      Decimal := Decimal + Valeurs[IndiceActuel]
   else
       (* situation du type IV ou IX ou XL, ... *)
      Decimal := Decimal - Valeurs[IndiceActuel];
   end;
end;
```

(1) Comme annoncé, les deux paramètres *Romain* et *Decimal* de DECIMALISER sont cités entre parenthèses après l'intitulé de la procédure. La signification de la mention var qualifiant le paramètre *Decimal* apparaîtra plus loin.

Il reste, pour en avoir fini avec DECIMALISER à proposer un contenu aux actions REMPLIR_VALEURS et TRANSFORMER.

Le remplissage du tableau *Valeurs* se fait de manière purement séquentielle. Je me contenterai donc de

1.6.5 REMPLIR_VALEURS. Comment dire ?

```
procedure REMPLIR_VALEURS;
       (* elle initialise le tableau Valeurs en y plaçant la valeur des divers chiffres romains : le
       tiroir I reçoit 1, V reçoit 5,...*)
begin
Valeurs[I]:=1;
Valeurs[V]:=5;
Valeurs[X]:=10;
Valeurs[L]:=50;
Valeurs[C]:=100;
Valeurs[D]:=500;
Valeurs[M]:=1000;
end;
```

Pascal offre d'autres manières d'initialiser un tableau : les constantes typées, qui sont en quelque sorte des variables initialisées dès la compilation. Elles permettent de raccourcir l'écriture proposée ci-dessus.

Le cas de TRANSFORMER est plus intéressant puisqu'il va nous permettre de découvrir une nouvelle forme de structure alternative : les choix multiples gérés pas la structure "Au cas où..." ("case ... of..." en Pascal)

1.6.6 TRANSFORMER. Comment faire faire ?

Il s'agit donc simplement, disposant du caractère contenu dans le paramètre *Caractere*, de placer dans la paramètre *Indice*, de type ChiffresRomains, l'indice correspondant.

On peut donc proposer une cascade d'alternatives :

```
si Caractere = 'I' alors
       Indice := I
sinon
       si Caractere = 'V' alors
              Indice := V
       sinon
```

```
si Caractere = 'X' alors
        Indice := X
sinon
        si Caractere = 'L' alors
                Indice := L
        sinon
                si Caractere = 'C' alors
                        Indice := C
                sinon
                        si Caractere = 'D' alors
                                Indice := D
                        sinon
                                Indice := M
```

On envisage donc successivement les valeurs 'I', 'V', 'X',... pour *Caractere* en plaçant à chaque fois les constantes de type ChiffresRomains correspondantes dans *Indice*.

Lorsqu'on a une cascade d'alternatives *de ce genre*, Pascal offre une structure particulière qui va faciliter l'écriture. Plutôt que

```
if Caractere = 'I' then
     Indice := I
else
     if Caractere = 'V' then
          Indice := V
     else
          if Caractere = 'X' then
               Indice := X
          else
               if Caractere = 'L' then
                    Indice := L
               else
                    if Caractere = 'C' then
                         Indice := C
                    else
                         if Caractere = 'D' then
                              Indice := D
                         else
                              Indice := M
```

qui, il faut bien le reconnaître, est passablement fastidieux, on pourra écrire :

```
case Caractere of
        'I' : Indice := I;
        'V' : Indice := V;
        'X' : Indice := X;
        'L' : Indice := L;
        'C' : Indice := C;
        'D' : Indice := D;
     else
          Indice := M;
end
```

Nous pourrions même ici en profiter pour introduire une vérification (très partielle et **non exigée** par les spécifications de TRANSFORMER) de l'écriture du nombre en chiffres romains en proposant :

```
case Caractere of
        'I' : Indice := I;
        'V' : Indice := V;
        'X' : Indice := X;
        'L' : Indice := L;
        'C' : Indice := C;
        'D' : Indice := D;
```

```
        'M' : Indice := M;
     else
        begin
        writeln('Nombre incorrect');
        halt; (* arrêt du programme global *)
        end;
end
```

Dès lors, si *Caractere* n'est pas un des chiffres romains, la procédure TRANSFORMER provoque un message d'erreur puis, à travers l'instruction halt, un arrêt du programme tout entier.

> Je n'en dirai pas davantage de cette instruction "halt" permettant l'arrêt "en catastrophe" d'un programme Pascal. Son utilisation ici ne correspond pas aux spécifications générales qui postulent que les nombres romains sont correctement écrits et ne prévoient donc aucune validation de ces nombres.

Dans le cours de l'écriture en pseudo-code nous représenterons la structure "Au cas où ..." de la manière suivante :

```
Au cas où Caractere vaut
   'I' :   Indice ← I
   'V' :   Indice ← V
   'X' :   Indice ← X
   'L' :   Indice ← L
   'C' :   Indice ← C
   'D' :   Indice ← D
   'M' :   Indice ← M
sinon
   Affiche 'Nombre incorrect'
   Halt
```

Nous reviendrons plus loin sur les possibilités et les contraintes liées à cette nouvelle structure.

Elle apporte en tout cas une solution immédiate à l'analyse de TRANSFORMER et permet de proposer :

1.6.7 TRANSFORMER. Comment dire ?

```
procedure TRANSFORMER(Caractere:char; var Indice:ChiffresRomains);
       (* elle fait transformer le paramètre valeur Caractere de type char en l'indice Indice
       (paramètre de type ChiffresRomains) *)
begin
case Caractere of
   'I' : Indice:=I;
   'X' : Indice:=X;
   'V' : Indice:=V;
   'L' : Indice:=L;
   'C' : Indice:=C;
   'D' : Indice:=D;
   'M' : Indice:=M
  else
   begin
   writeln('Erreur dans le nombre en chiffres romains. Tapez Entrée');
   readln; (* pour avoir le temps de lire le message avant l'arrêt du programme *)
   Halt;
   end;
end;
end;
```

Et nous en avons dès lors terminé avec la procédure DECIMALISER. Il nous reste donc à envisager l'autre face du problème : l'analyse de la procédure ROMANISER qui va transformer le

nombre habituel *Decimal* (paramètre de type entier long de ROMANISER) en son équivalent romain *Romain* (paramètre de ROMANISER, de type NombreRomain).

1.7 Analyse de ROMANISER

1.7.1 ROMANISER. Quoi faire ?

On relira d'abord avec attention les spécifications de l'action ROMANISER, page 126.

1.7.2 ROMANISER. Comment faire ?

Il s'agit donc de trouver une stratégie qui sur base d'un nombre entier écrit comme nous le faisons aujourd'hui, lui associe son équivalent en chiffres romains.

La première règle à mettre en oeuvre est celle qui oblige à une écriture en deux parties lorsque le nombre est supérieur ou égal à 4000, comme décrit à la page 121.

Si donc le nombre *Decimal* est supérieur ou égal à 4000, on en isolera le nombre de milliers (= la partie multipliant 1000) et la partie inférieure à 1000. On appliquera ensuite à ces deux parties pratiquement le même traitement : leur écriture en chiffres romains, à la seule différence que l'éventuelle partie multiple de 1000 (= le nombre de milliers de *Decimal*) aura dans son équivalent en chiffres romains tous ses symboles assortis de °.

Pour obtenir le nombre de milliers de *Decimal*, il suffit d'effectuer la division entière de *Decimal* par 1000 et pour obtenir la portion de *Decimal* inférieure à 1000, de prendre le reste de la division entière de *Decimal* par 1000.

1.7.3 ROMANISER. Comment faire faire ?

On pourrait d'abord écrire en utilisant les deux variables *RomainPlusGrandQue1000* et *RomainPlusPetitQue1000,* toutes deux de type NombreRomain :

```
Si Decimal ≥ 4000 alors
    [ROMANISER_MORCEAU_SUPERIEUR_A_1000]        RomainPlusGrandQue1000 de type NombreRomain
    [ROMANISER_MORCEAU_INFERIEUR_A_1000]        RomainPlusPetitQue1000 de type NombreRomain
    Romain ← RomainPlusGrandQue1000 + RomainPluspetitQue1000 (1)
sinon
    [ROMANISER_DECIMAL]
```

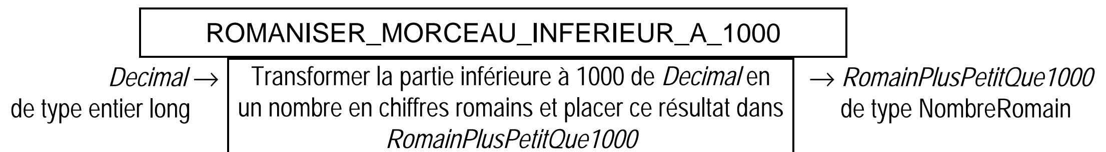
(1) Le signe + dénote ici la concaténation (= le fait de les coller l'une à l'autre) des deux chaînes de caractères *RomainPlusGrandQue1000* et *RomainPluspetitQue1000*. La première de ces variables *RomainPlusGrandQue1000* sert à contenir l'équivalent en chiffres romains du nombre de milliers de *Decimal*; la seconde *RomainPluspetitQue1000* sert à contenir l'équivalent en chiffres romains de la partie de *Decimal* inférieure à 1000.

Pour mieux saisir ce que représentent les deux variables *RomainPlusGrandQue1000* et *RomainPluspetitQue1000,* il reste comme d'habitude à préciser les actions complexes apparues :

avec

Avant	Après
- *Decimal* de type entier long, contenant un entier entre 4000 et 998001	- *RomainPlusGrandQue1000,* de type NombreRomain contenant l'équivalent en chiffres romains assortis du symbole ° de la partie de *Decimal* multiple de 1000.

et

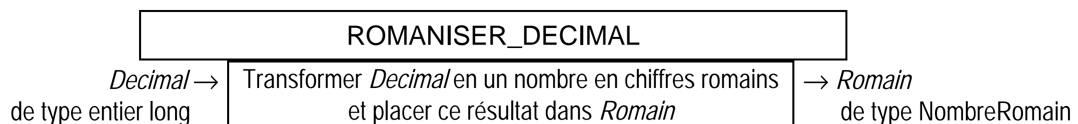


avec

Avant	Après
- *Decimal* de type entier long, contenant un entier entre 4000 et 998001	- *RomainPlusPetitQue1000,* de type NombreRomain contenant l'équivalent en chiffres romains de la partie de *Decimal* strictement inférieure à 1000.

On notera que pour chacune de ces actions, le paramètre Decimal contient un nombre supérieur à 4000, étant donné l'analyse de ROMANISER telle que conduite ci-dessus.

Et enfin :



avec

Avant	Après
- *Decimal* de type entier long, contenant un entier entre 0 et 3999	- *Romain,* de type NombreRomain contenant l'équivalent en chiffres romains de *Decimal.*

Une fois de plus, on peut unifier en une action unique ces diverses actions, même si l'écriture du symbole ° pour la partie de *Decimal* multiple de 1000 complique un peu la tâche.

Plutôt que de passer *Decimal* à ROMANISER_MORCEAU_SUPERIEUR_A_1000 en confiant à cette procédure le fait d'extraire la partie multipliant 1000 de *Decimal* et de la "romaniser" et de passer la même valeur *Decimal* à ROMANISER_MORCEAU_ INFERIEUR_A_1000 en demandant à cette procédure d'extraire la partie inférieure à 1000 et la "romaniser", nous allons en tout cas, lorsque *Decimal* est supérieur à 4000, commencer par isoler la partie de *Decimal* multiple de 1000, puis la partie inférieure à 1000 avant d'appeler à deux reprises une procédure unique de "romanisation" de ces deux "morceaux".

Il est facile de voir que cette action, que nous allons baptiser ROMANISER_TOUT_OU_PARTIE, doit en tout cas être assortie d'un paramètre "d'entrée" dans lequel nous passerons d'abord *Decimal* div 1000 (partie de *Decimal* multipliant 1000), puis *Decimal* mod 1000 (partie de *Decimal* inférieure à 1000) ou encore *Decimal* lui-même. Il nous faut aussi un paramètre de "sortie", baptisé *Romain*, qui sera, dans le premier cas un "prête-nom" pour *NombreRomainPlusGrandQue1000*, dans le second cas un "prête-nom" pour *NombreRomainPlusPetitQue1000*, et enfin dans le troisième cas un "prête-nom" pour *Romain* lui-même.

Il nous faut cependant un paramètre d'entrée supplémentaire qui signale si oui ou non la procédure ROMANISER_TOUT_OU_PARTIE reçoit la partie de *Decimal* multiple de 1000 (auquel cas il faut adjoindre aux caractères composant le nombre en chiffres romains obtenu, le symbole °).

Ainsi donc, l'action ROMANISER_TOUT_OU_PARTIE sera accompagnée de 2 paramètres d'entrée :

- *Habituel*, de type entier long, qui recevra le nombre décimal à transformer en nombre romain
- *AuDelaDe1000*, booléen, vrai lorsque le nombre passé dans *Habituel* constitue la partie multiple de 1000 de *Decimal* et que dès lors les caractères du résultat doivent être accompagnés de °.

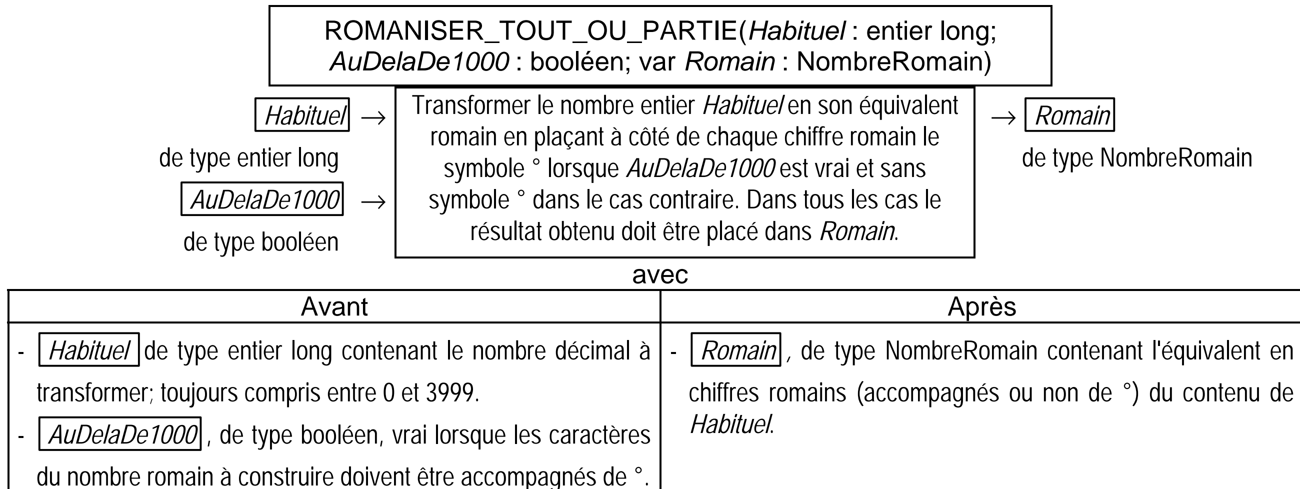
  En effet, en l'absence d'un paramètre *AuDelaDe1000* signalant s'il s'agit de transformer en chiffres romains la partie de *Decimal* multipliant 1000 ou la partie inférieure à 1000 ou encore *Decimal* lui-même, la procédure ROMANISER_TOUT_OU_PARTIE ne "connaîtrait" et ne travaillerait que sur la valeur donnée au paramètre *Habituel* sans "savoir" si la "romanisation" doit ou non accompagner les caractères romains du symbole °. Le simple fait de garnir le paramètre *Habituel* avec la valeur *Decimal* div 1000, avec *Decimal* mod 1000 ou avec *Decimal*, ne suffit pas pour que ROMANISER_TOUT_OU_PARTIE "sache" dans quel cas on se trouve. Ainsi ROMANISER_TOUT_OU_PARTIE recevant 68 comme valeur de *Habituel* ne peut savoir si 68 est la partie du nombre à transformer multipliant 1000 (auquel cas le symbole ° doit être adjoint aux chiffres romains) ou la partie inférieure à 1000 (auquel cas le symbole ° est inutile). Il faut donc un paramètre supplémentaire *AuDelaDe1000* sur base duquel décider si ° doit ou non être adjoint.

  Tout cela apparaîtra plus nettement encore lorsque je détaillerai (page 148) la manière dont s'effectuera le dialogue au moment d'un appel de procédure assortie de paramètres.

Il faut également un paramètre de sortie :

- *Romain* de type NombreRomain comportant l'équivalent en chiffres romains du nombre reçu dans *Habituel*.

En résumé, nous avons donc :



avec

Avant	Après
- *Habituel* de type entier long contenant le nombre décimal à transformer; toujours compris entre 0 et 3999. - *AuDelaDe1000*, de type booléen, vrai lorsque les caractères du nombre romain à construire doivent être accompagnés de °.	- *Romain*, de type NombreRomain contenant l'équivalent en chiffres romains (accompagnés ou non de °) du contenu de *Habituel*.

Avec cette description de l'action de "romanisation", on peut réécrire la marche à suivre correspondant à ROMANISER :

Si Décimal ≥ 4000 alors

  ROMANISER_TOUT_OU_PARTIE (*Decimal* div 1000, vrai, *RomainPlusGrandQue1000*)

  RomainPlusGrandQue1000 de type NombreRomain

  ROMANISER_TOUT_OU_PARTIE (*Decimal* mod 1000, faux, *RomainPlusPetitQue1000*)

  RomainPlusPetitQue1000 de type NombreRomain

  Romain ← *RomainPlusGrandQue1000* + *RomainPluspetitQue1000*

sinon

  ROMANISER_TOUT_OU_PARTIE (*Decimal*, faux, *Romain*)

Il faut une fois de plus mettre en parallèle la définition des paramètres de ROMANISER_TOUT_OU_PARTIE et ce qui leur est associé lors de chaque appel :

```
Définition:  ROMANISER_TOUT_OU_PARTIE(       Habituel;  AuDelaDe1000;       Romain)
1er appel:   ROMANISER_TOUT_OU_PARTIE(Decimal div 1000, vrai, RomainPlusGrandQue1000)
Définition:  ROMANISER_TOUT_OU_PARTIE(       Habituel;  AuDelaDe1000;       Romain)
2è appel:    ROMANISER_TOUT_OU_PARTIE(Decimal mod 1000, faux, RomainPlusPetitQue1000)
Définition:  ROMANISER_TOUT_OU_PARTIE(       Habituel;  AuDelaDe1000;       Romain)
3è appel :   ROMANISER_TOUT_OU_PARTIE(       Decimal,     faux,             Romain)
```

Autrement dit, au lieu de

- ROMANISER_MORCEAU_SUPERIEUR_A_1000, on appellera ROMANISER_TOUT_OU_PARTIE en précisant que
  - *Habituel* vaut *Decimal* div 1000
  - *AuDelaDe1000* est vrai
  - *Romain* est *RomainPlusGrandQue1000*

au lieu de

- ROMANISER_MORCEAU_INFERIEUR_A_1000, on appellera ROMANISER_TOUT_OU_PARTIE en précisant que
  - *Habituel* vaut *Decimal* mod 1000
  - *AuDelaDe1000* est faux
  - *Romain* est *RomainPlusPetitQue1000*

et enfin, au lieu de

- ROMANISER_DECIMAL, on appellera ROMANISER_TOUT_OU_PARTIE en précisant que
  - *Habituel* vaut *Decimal*
  - *AuDelaDe1000* est faux
  - *Romain* est *Romain*

Nous sommes dès lors, à ce stade, en mesure de répondre à

1.7.4 ROMANISER. Comment dire ?

```
procedure ROMANISER(Decimal: LongInt; var Romain:NombreRomain);
       (* Elle fait transformer le nombre entier Decimal (paramètre valeur) en une chaîne contenant
       ce nombre en chiffres romains : Romain (de type NombreRomain) *)

var RomainPlusGrandQue1000,
    RomainPlusPetitQue1000
       (* pour accueillir les 2 chaînes représentant en chiffres romains, la partie multiple de 1000
       et celle plus petite que 1000 *)
       : NombreRomain;

procedure ROMANISER_TOUT_OU_PARTIE(Habituel : LongInt;
       AuDelaDe1000:boolean; var Romain : NombreRomain);
       (* Elle fait transformer le nombre décimal Habituel (compris entre 0 et 3999) en son
       équivalent en chiffres romains soit Romain. Lorsque le paramètre AuDelaDe1000 est vrai, les
       chiffres romains sont assortis du symbole ° pour rappeler qu'ils multiplient 1000 *)

begin (* début de ROMANISER *)
if Decimal >= 4000 then
   begin
```

```
    ROMANISER_TOUT_OU_PARTIE(Decimal div 1000, true, RomainPlusGrandQue1000);
    ROMANISER_TOUT_OU_PARTIE(Decimal mod 1000, false, RomainPlusPetitQue1000);
        (* concaténation des 2 chaînes correspondant à la partie multipliant 1000 et celle plus
        petite que 1000 *)
    Romain := RomainPlusGrandQue1000 + RomainPlusPetitQue1000;
    end
else
    ROMANISER_TOUT_OU_PARTIE(Decimal,false,Romain);
end;
```

Il nous reste cependant à analyser ROMANISER_TOUT_OU_PARTIE qui transformera le nombre décimal *Habituel*, plus grand ou égal à 0 et strictement inférieur à 4000, en son équivalent *Romain*.

1.8 Analyse de ROMANISER_TOUT_OU_PARTIE

1.8.1 ROMANISER_TOUT_OU_PARTIE. Quoi faire ?

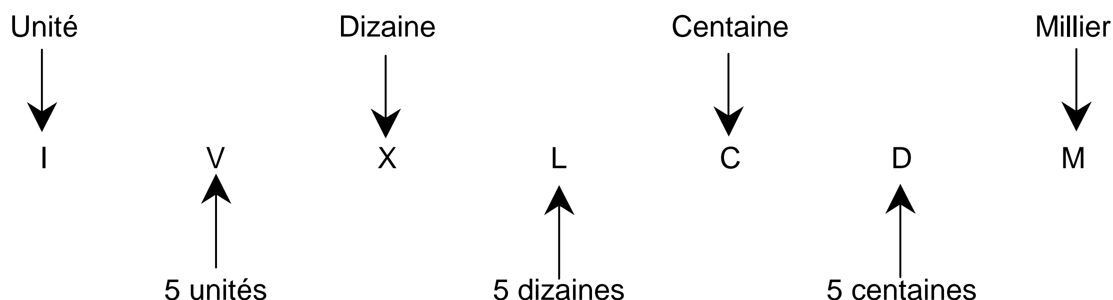
Les spécifications sont indiquées à la page 140.

1.8.2 ROMANISER_TOUT_OU_PARTIE. Comment faire ?

Disposant d'un nombre décimal, par exemple 3698, il nous faut mettre au jour une stratégie d'écriture en chiffres romains.

Nous allons successivement envisager, de la gauche vers la droite, le chiffre des milliers (ici 3), puis celui des centaines (ici 6), celui des dizaines (9) et enfin celui des unités (8). Cependant, pour pouvoir écrire, pour chaque chiffre, son équivalent en symboles romains, il nous faut à la fois savoir quel est ce chiffre, mais aussi quel est son rang. Par exemple face au chiffre 6, il nous faut également savoir s'il s'agit de centaines, auquel cas la traduction est DC, s'il s'agit de dizaines, ce qui donnerait LX ou d'unités ce qui conduirait à VI.

Nous basons donc notre écriture sur une concordance :



Avec ce tableau devant les yeux, notre stratégie pourrait être :

- On parcourt le nombre *Habituel* de gauche à droite, pour disposer successivement du chiffre des milliers, puis des centaines,..., symbole après symbole
- A chaque chiffre obtenu :
    - on évalue d'après son rang, sur base du tableau ci-dessus, le symbole correspondant
    - sur base de la valeur du chiffre et connaissant le symbole à écrire, on écrit à droite de ce qu'on a déjà obtenu

      1 fois le symbole pour la valeur 1

2 fois le symbole pour la valeur 2
3 fois le symbole pour la valeur 3
le symbole suivi du symbole qui est à sa droite pour la valeur 4
le symbole qui est à sa droite pour la valeur 5
le symbole qui est à sa droite, suivi du symbole pour la valeur 6
le symbole qui est à sa droite, suivi de 2 fois le symbole pour la valeur 7
le symbole qui est à sa droite, suivi de 3 fois le symbole pour la valeur 8
le symbole suivi du symbole qui est de 2 positions à sa droite pour la valeur 9

- On a obtenu ainsi la valeur de *Romain*

Ainsi, pour 3698, on aura successivement :

- Chiffre 3 : rang : millier, valeur : 3, donc
  - comme le rang est celui des milliers, le symbole est M;
  - comme la valeur est 3, on écrit MMM.
- Chiffre 6 : rang : centaine, valeur : 6, donc
  - comme le rang est celui des centaines, le symbole est C;
  - comme la valeur est 6, on écrit le symbole qui est à sa droite, soit D, suivi du symbole, soit au total, on ajoute à droite de ce qui est écrit DC.
- Chiffre 9 : rang : dizaine, valeur : 9, donc
  - comme le rang est celui des dizaines, le symbole est X;
  - comme la valeur est 9, on écrit le symbole soit X suivi du symbole qui est de 2 positions à sa droite, soit C; donc au total, on ajoute à droite de ce qui est déjà écrit XC.
- Chiffre 8 : rang : unité, valeur : 8, donc
  - comme le rang est celui des unités, le symbole est I;
  - comme la valeur est 8, on écrit le symbole qui est à sa droite, soit V, suivi de 3 exemplaires du symbole, soit au total, on ajoute à droite de ce qui est écrit VIII.

On obtient donc : MMMDCXCVIII.

Remarquons qu'évidemment, on pourrait aussi parcourir le nombre de droite à gauche (des unités vers les milliers), que le tableau de concordances représenté plus haut aurait également pu être inversé (de M à I, plutôt que de I à M). Dans tous les cas la stratégie reste semblable.

Il resterait à prendre en compte l'écriture du symbole ° après chaque chiffre romain, lorsque le paramètre *AuDelaDe1000* est vrai, mais cela c'est immédiat.

1.8.3 ROMANISER_TOUT_OU_PARTIE. Comment faire faire ?

Il nous reste à formaliser la stratégie décrite, après avoir mis en évidence le tableau qui nous sera indispensable.

1.8.3.1 Structure de données : le tableau nécessaire

Il doit rendre compte des concordances présentées par le tableau décrit ci-dessus :

unité → I et 5 unités → V
dizaine → X et 5 dizaines → L
centaine → C et 5 centaines → D
millier → M.

On pourrait penser dans un premier temps à un tableau similaire à celui déjà employé

Valeurs

1	5	10	50	100	500	1000
I	V	X	L	C	D	M

Malheureusement, si ce tableau permettait d'établir fort rapidement une correspondance chiffre romain → valeur décimale, il ne permet pas aisément d'établir la correspondance inverse, et c'est cette dernière qui nous est indispensable.

> Il faut remarquer qu'en général en ce qui concerne les tableaux, déterminer, sur base d'une étiquette, le contenu du tiroir concerné est un jeu d'enfant : c'est à cela que servent les [ ]. La correspondance inverse est nettement plus malaisée : connaissant un contenu possible de tiroir, déterminer l'étiquette correspondante demande une exploration de tous les tiroirs jusqu'à ce qu'on ait trouvé "le bon".
>
> Si, face à une étagère, je vous demande ce que contient le tiroir marqué 4, il vous suffit d'ouvrir ce dernier pour me répondre. Si je vous affirme que l'un des tiroirs contient (peut-être) un trésor et que je vous demande lequel (= quelle est l'étiquette du tiroir au trésor), il va vous falloir les ouvrir tous, l'un après l'autre, jusqu'à avoir trouvé le bon.

Ce qu'il faudrait donc c'est en quelque sorte, dans le tableau ci-dessus, inverser le rôle des étiquettes et des contenus de tiroirs :

Valeurs

I	V	X	L	C	D	M
1	5	10	50	100	500	1000

et même, puisque les contenus des tiroirs peuvent bien entendu être des caractères :

Valeurs

'I'	'V'	'X'	'L'	'C'	'D'	'M'
1	5	10	50	100	500	1000

Mais, une fois de plus, les étiquettes ne sont pas adéquates : elles ne constituent pas un intervalle dans les entiers, mais quelques entiers épars : 1, 5, 10, 50, ...

Ce qu'il faudrait, c'est des étiquettes convenables, mais qui soient en correspondance avec 1, 5, 10, 50,... Cette correspondance est immédiate (même si elle peut paraître à certains, qui n'ont plus que de lointains souvenirs d'arithmétique, un peu "tirée par les cheveux").

unité → 1 $= 10^{0} = 10^{\mathbf{0}/2}$ étiquette : 0 contenu du tiroir : 'I'
5 unités étiquette : 1 contenu du tiroir : 'V'
dizaine → 10 $= 10^{1} = 10^{\mathbf{2}/2}$ étiquette : 2 contenu du tiroir : 'X'
5 dizaines étiquette : 3 contenu du tiroir : 'L'
centaine → 100 $= 10^{2} = 10^{\mathbf{4}/2}$ étiquette : 4 contenu du tiroir : 'C'
5 centaines étiquette : 5 contenu du tiroir : 'D'
millier → 1000 $= 10^{3} = 10^{\mathbf{6}/2}$ étiquette : 6 contenu du tiroir : 'M'

Le tableau utilisé serait donc finalement :

V

'I'	'V'	'X'	'L'	'C'	'D'	'M'
0	1	2	3	4	5	6

ce qui se traduira en Pascal par : `V : array[0..6] of char`

Mais il faudra donc "jouer" avec la correspondance entre étiquettes et puissances de 10, comme ci-dessus.

1.8.3.2 Marche à suivre

```
V[0] ← 'I'    V[1] ← 'V'    V[2] ← 'X'    V[3] ← 'L'
V[4] ← 'C'    V[5] ← 'D'    V[6] ← 'M'    (1)
Romain ← ' ' (2)
Si AuDelaDe1000 alors
        Signe ← '°'                                          Signe de type string[1]
sinon
        Signe ← ' ' (2)
C ← 6                                                        C de type intervalle entier -2..6 (9)
Tant que C ≥ 0
        [CALCULER_10_EXP_C_DIV_2] et le placer dans Expo     Expo de type integer
        N ← Habituel div Expo (3)                            N de type integer
        Habituel ← Habituel mod Expo (4)
        Au cas où N vaut
                1 :   Romain ← Romain + V[C] + Signe (5)
                2 :   Romain ← Romain + V[C] + Signe + V[C] + Signe
                3 :   Romain ← Romain + V[C] + Signe + V[C] + Signe + V[C] + Signe
                4 :   Romain ← Romain + V[C] + Signe + V[C+1] + Signe  (6)
                5 :   Romain ← Romain + V[C+1] + Signe
                6 :   Romain ← Romain + V[C+1] + Signe + V[C] + Signe
                7 :   Romain ← Romain + V[C+1] + Signe + V[C] + Signe + V[C] + Signe
                8 :   Romain ← Romain + V[C+1] + Signe + V[C] + Signe + V[C] + Signe + V[C] + Signe
                9 :   Romain ← Romain + V[C] + Signe + V[C+2] + Signe  (7)
         C ← C - 2 (8)
```

Le coeur de cette marche à suivre, c'est une boucle, commandée par un compteur *C* qui passera par les valeurs 6, 4, 2, puis 0 (-2 provoquant l'arrêt). A chaque passage, la valeur de *C* donnera d'une part une puissance de 10 (en réalité $10^{C \text{ div } 2}$ donc 10^3, puis 10^2, puis 10^1, puis 10^0) et d'autre part, en consultant le tiroir d'étiquette *C* de *V*, le symbole romain correspondant.

Quelques remarques supplémentaires :

(1) C'est par souci d'économie de place que les actions d'affectation destinées à garnir le tableau *V* sont placés côte à côte plutôt que l'une en dessous de l'autre.

(2) D'une part, le paramètre *Romain* qui finira par contenir le nombre romain à obtenir, est initialisé par une chaîne vide; d'autre part, c'est également la chaîne vide qui est placée dans *Signe* lorsque *AuDelaDe1000* est faux.

(3) *Expo* contenant successivement 10^3, puis 10^2 puis 10^1, puis 10^0, N contiendra successivement le chiffre des milliers, puis, au tour suivant de la boucle Tant que ..., celui des centaines, puis celui des dizaines et enfin celui des unités.

(4) Conjointement, *Habituel*, qui contenait primitivement le nombre à transformer, contiendra les restes successifs de la division par les mêmes puissances de 10.

Ainsi, si *Habituel* valait primitivement 3698, on aura successivement :

Expo : 1000   *N* : 3   *Habituel* : 698
Expo : 100   *N* : 6   *Habituel* : 98
Expo : 10   *N* : 9   *Habituel* : 8
Expo : 1   *N* : 8   *Habituel* : 0

Semblablement, si *Habituel* valait primitivement 302, on aura successivement :

Expo : 1000 *N* : 0 *Habituel* : 302
Expo : 100 *N* : 3 *Habituel* : 2
Expo : 10 *N* : 0 *Habituel* : 2
Expo : 1 *N* : 2 *Habituel* : 0

(5) *V*[*C*] est le symbole romain à adjoindre à droite de ce que *Romain* contient déjà. Il est suivi de *Signe* (qui contient soit ° soit la chaîne vide).

(6) Lorsque *N*, contenant le chiffre concerné, vaut 4, les symboles à adjoindre à *Romain*, sont, non pas 4 fois *V*[*C*], mais *V*[*C*] suivi de *V*[*C*+1] (comme dans IV ou XL ou CD).

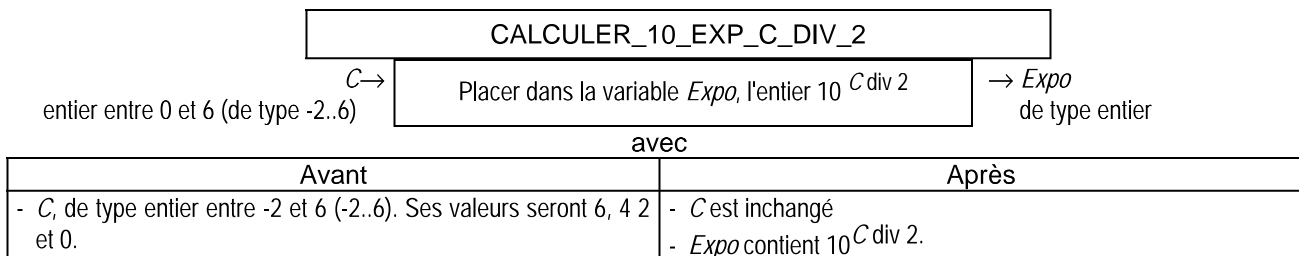
(7) Semblablement, lorsque *N*, contenant le chiffre concerné, vaut 9, les symboles à adjoindre à Romain, sont, non pas 9 fois *V*[*C*], mais *V*[*C*] suivi de *V*[*C*+2] (comme dans IX ou XC ou CM).

(8) Le compteur gouvernant la boucle ne peut passer que par les valeurs 6, 4, 2 et 0 (et -2 qui provoquera l'arrêt). C'est pour cette raison qu'on écrit *C* - 2 et qu'on n'a pas utilisé une boucle "Pour...".

(9) Il faut noter que lorsque la boucle Tant que... s'interrompra, c'est parce que *C* aura pris la valeur -2. Si on souhaite donner à *C* une valeur de type intervalle, il doit donc s'agir de l'intervalle -2..6 et non 0..6.

Il faut remarquer aussi que si *Habituel* est nul, *N* reste constamment nul dans chaque itération de la boucle Tant que... et aucune des alternatives rencontrées dans la structure Au cas où... n'est rencontrée. *Romain* reste donc vide (comme primitivement défini).

Il reste, pour être complet, à spécifier l'action complexe CALCULER_10_EXP_C_DIV_2 :



Avant	Après
- *C*, de type entier entre -2 et 6 (-2..6). Ses valeurs seront 6, 4 2 et 0.	- *C* est inchangé - *Expo* contient $10^{C \text{ div } 2}$.

Nous voici donc en mesure d'écrire le texte de la procédure ROMANISER_TOUT_OU_PARTIE.

1.8.4 ROMANISER_TOUT_OU_PARTIE. Comment dire ?

```
procedure ROMANISER_TOUT_OU_PARTIE(Habituel : LongInt;
          AuDelaDe1000:boolean; var Romain : NombreRomain);
(* Elle fait transformer le nombre décimal Habituel (compris entre 0 et 3999) en son équivalent en
chiffres romains soit Romain. Lorsque le paramètre AuDelaDe1000 est vrai, les chiffres romains sont
assortis du symbole ° pour rappeler qu'ils multiplient 1000 *)

var V : array[0..6] of char;
          (* la composante 0 contiendra le caractère I, la composante 1 le caractère V,...*)
      Signe : string[1];
          (* qui d'après la valeur de AuDelaDe1000, sera '°' ou la chaîne vide *)
      C : -2..6;
          (* compteur pour le "parcours" de Habituel *)
      N,
          (* qui contiendra les quotients des divisions de Habituel par 1000, 100,... *)
      Expo
          (* qui contiendra les puissances de 10 intervenant dans les divisions de Habituel *)
          (* ON NE CONNAIT PAS ENCORE ICI LE CONCEPT DE FONCTION *)
          :integer;
```

```
procedure CALCULER_10_EXP_C_DIV_2;
(* Elle fournit dans la variable Expo la valeur 10 exposant(C div 2) *)

begin (* début de ROMANISER_TOUT_OU_PARTIE *)
V[0]:='I'; V[1]:='V'; V[2]:='X'; V[3]:='L';
V[4]:='C'; V[5]:='D'; V[6]:='M';
(* au début la chaîne Romain doit être vide : on lui adjoindra divers caractères au fur et à mesure
de l'exploration de Habituel *)
Romain:='';
(* choix du symbole à faire figurer auprès des chiffres romains (normaux ou à multiplier par 1000,
lorsqu'ils sont accompagnés de ° *)
if AuDelaDe1000 then
   Signe:='°'
else
   Signe:='';
C:=6;
while C>=0 do
   begin
   CALCULER_10_EXP_C_DIV_2;
   (* Expo contient 10 exposant (C div 2) : 1000, puis 100, ... puis 1 *)
   N:=Habituel div Expo; (* N sera le chiffre des milliers, puis des centaines, ... *)
   Habituel:=Habituel mod Expo;
   case N of
      1 : Romain:=Romain+V[C]+Signe;
      2 : Romain:=Romain+V[C]+Signe+V[C]+Signe;
      3 : Romain:=Romain+V[C]+Signe+V[C]+Signe++V[C]+Signe;
      4 : Romain:=Romain+V[C]+Signe+V[C+1]+Signe;
      5 : Romain:=Romain+V[C+1]+Signe;
      6 : Romain:=Romain+V[C+1]+Signe+V[C]+Signe;
      7 : Romain:=Romain+V[C+1]+Signe+V[C]+Signe+V[C]+Signe;
      8 : Romain:= Romain+V[C+1]+Signe+ V[C]+Signe+V[C]+ Signe+ V[C]+Signe;
      9 : Romain:=Romain+V[C]+Signe+V[C+2]+Signe;
   end;
   C:=C-2;
   end;
end;
```

1.9 Analyse de CALCULER_10_EXP_C_DIV_2

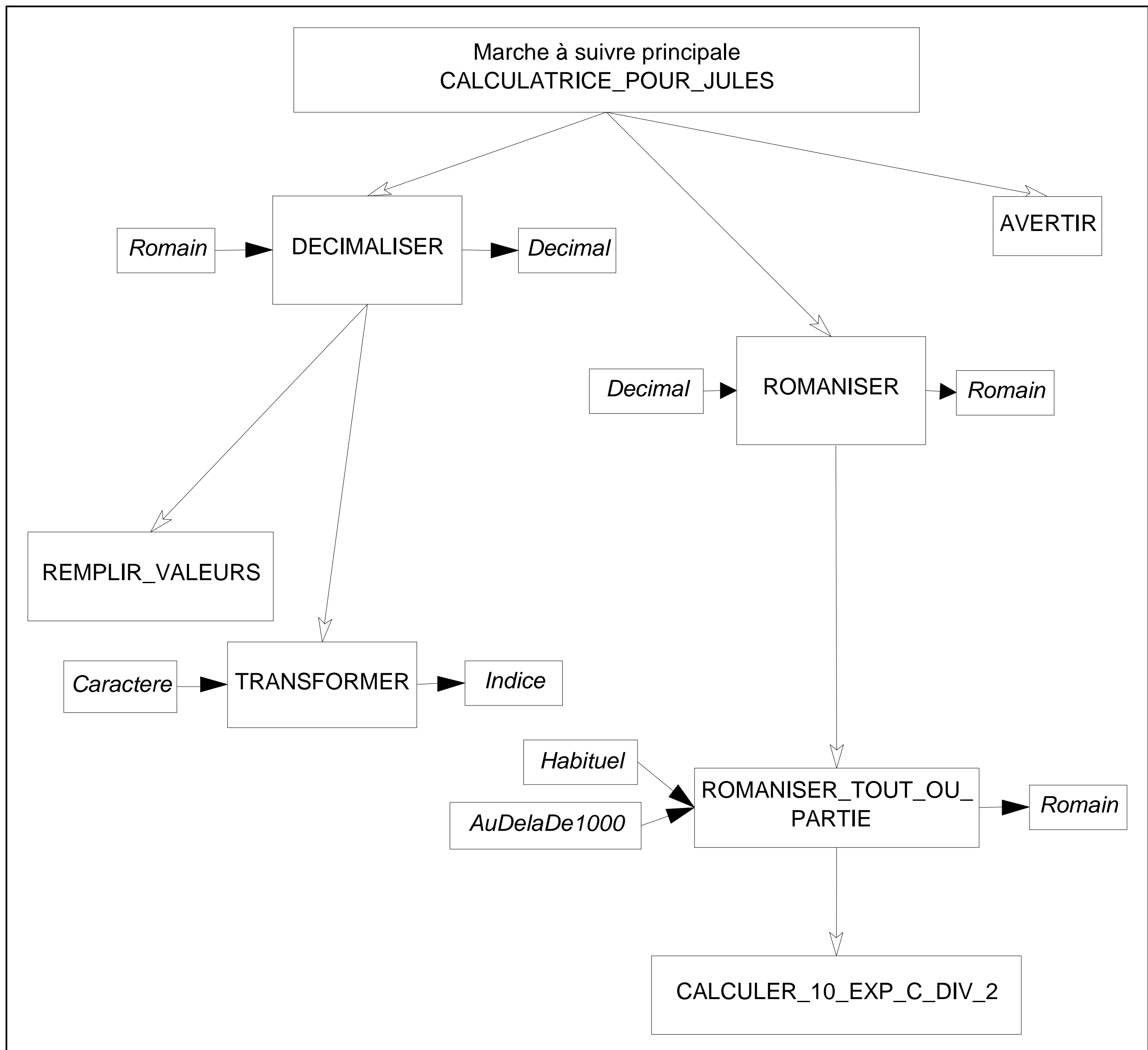
C'est essentiellement parce que la fonction puissance (ou exposant) n'existe pas en Pascal que nous sommes contraints d'écrire cette procédure ad-hoc.

Je me contenterai cependant de fournir le programme Pascal correspondant; le tour de main est bien connu : on va boucler en multipliant une quantité (primitivement initialisée à 1) à chaque tour de boucle.

1.9.1 CALCULER_10_EXP_C_DIV_2. Comment dire

```
procedure CALCULER_10_EXP_C_DIV_2;
(* Elle fournit dans la variable Expo la valeur 10^(C div 2) *)
var Co : integer;

begin
Expo:=1;
for Co:=1 to C div 2 do
   Expo:=Expo*10;
end;
```

Et ceci termine toute l'analyse menée. La structure du programme peut se résumer par le schéma suivant :



Les procédures assorties de paramètres sont représentées en grisé et les paramètres les accompagnant sont clairement indiqués.

1.10 Et à l'exécution ? : les paramètres

Il me faut à présent ajouter quelques informations essentielles sur la manière dont sont gérés les paramètres lors des appels de procédures.

> Ce qui va suivre nécessite une grande attention et demande au lecteur de garder un oeil sur le texte du programme (particulièrement sur les instructions d'appel de procédure) et l'autre oeil sur les schémas qui vont s'enchaîner.

1.10.1 Début du travail de l'exécutant principal et de son installateur d'étiquettes

Comme j'ai déjà eu l'occasion de le préciser, entrent d'abord en scène la paire "exécutant principal" - "installateur d'étiquettes". Ce dernier dispose de la partie déclaration du programme principal :

 © CeFIS - FUNDP 2002

```
type NombreRomain=string[40];

var PremierNombreRomain, SecondNombreRomain, SommeRomain, DifferenceRomain, ProduitRomain
      : NombreRomain;
    PremierNombreDecimal, SecondNombreDecimal, SommeDecimal, DifferenceDecimal,
      ProduitDecimal : Longint;
    Reponse:char;
```

Dès lors, l'installateur dispose l'aide mémoire reprenant la déclaration de type (il n'y a pas ici de déclaration de constante) dans un coin du local :

NombreRomain=string[40]

puis il installe sur les casiers de type adéquat les étiquettes demandées. Le contenu de ces casiers est bien entendu quelconque (ce que je repésente ici en y plaçant le symbole ?)..

PremierNombreRomain	SecondNombreRomain	Reponse
'??????'	'??????'	'?'
PremierNombreDecimal	SecondNombreDecimal	
????	????	
DifferenceRomain	ProduitRomain	SommeRomain
'??????'	'??????'	'??????'
DifferenceDecimal	ProduitDecimal	SommeDecimal
????	????	????

A l'issue de cette installation, l'exécutant principal se met au travail, en suivant les indications de la marche à suivre principale (Cf. pages 126 ou 127). Il va donc lire les deux nombres, appeler à deux reprises un exécutant auxiliaire chargé de DECIMALISER ces 2 nombres, calculer la somme, la différence et le produit décimaux.

Nous le retrouvons, si vous voulez bien, au moment du premier appel de ROMANISER :

```
ROMANISER(SommeDecimal,SommeRomain);
```

en supposant qu'à l'issue du travail déjà fait le local ressemble à :

NombreRomain=string[40]

PremierNombreRomain	SecondNombreRomain	Reponse
'CMXIX'	**'DCCCXIV'**	'?'
PremierNombreDecimal	SecondNombreDecimal	
919	***814***	
DifferenceRomain	ProduitRomain	SommeRomain
'??????'	'??????	'??????'
DifferenceDecimal	ProduitDecimal	SommeDecimal
105	***748066***	***1733***

On y a noté en gras italique, dans les casiers concernés, les données pertinentes. Les autres casiers, dans lesquels les données sont celles qui y traînent par hasard, ont leur contenu remplis par des ????.

Pour ceux qui disposent du texte en couleur : dans les schémas qui suivront les étiquettes et les données manipulées par l'exécutant principal sont en noir, celles de ROMANISER en rouge, celles de ROMANISER_TOUT_OU_PARTIE en bleu et celles de CALCULER_10_EXP_C_DIV_2 en vert.

1.10.2 Le premier appel de l'exécutant chargé de ROMANISER

Comme nous le savons déjà, face à cet appel de procédure, l'exécutant principal va donc se tourner vers un couple exécutant - installateur d'étiquettes, chargé de ROMANISER. Notons, cependant que, par rapport à ce qui nous était habituel le mot ROMANISER est cette fois suivi par des indications entre parenthèses (correspondant à l'existence des paramètres). Dès, lors l'exécutant principal **ne quitte pas** comme d'habitude, c'est à dire lorsqu'il n'y a pas de paramètres, **la scène** au moment où il appelle le couple chargé de ROMANISER : avant de céder la place, un dialogue va avoir lieu; c'est ce dialogue que je vais à présent illustrer.

Le couple chargé de ROMANISER et qui s'avance à l'appel de l'exécutant principal dispose, nous le savons, du texte de la procédure correspondante (Cf. page 140 ou 141). L'installateur d'étiquettes de ROMANISER dispose en tout cas de la partie déclaration :

```
procedure ROMANISER(Decimal: LongInt; var Romain:NombreRomain);

var RomainPlusGrandQue1000, RomainPlusPetitQue1000 : NombreRomain;
```

C'est le moment de bien garder sous les yeux le schéma de la page 151.

Le couple trouve bien évidemment le local comme le laisse l'exécutant principal qui vient de faire appel à ses services (Cf. ci-dessus). L'installateur d'étiquettes, constatant l'existence d'une liste de paramètres, entame le dialogue suivant :

IE (L'installateur d'étiquettes de ROMANISER) : "J'ai un premier paramètre *Decimal*, de type LongInt (entier long) et de genre **valeur**[13]. Collons donc l'étiquette *Decimal* sur un casier vierge, comme je le fais d'habitude pour les variables locales."

(Il colle l'étiquette *Decimal* sur un casier vierge, de type entier long. Il se tourne alors vers l'exécutant principal :)

IE : "C'est à toi, exécutant principal qui vient de nous appeler, de me dire quelle donnée je dois placer dans le casier sur lequel je viens de mettre l'étiquette *Decimal*, correspondant à ce paramètre de genre valeur. Alors, que faut-il y mettre ?"

(L'exécutant principal, qui dispose, lui de l'instruction d'appel `ROMANISER(SommeDecimal, SommeRomain)` lui répond alors :)

EP : "Il faut y copier la donnée désignée par *SommeDecimal*, autrement dit, 1733, qui est le contenu du casier *SommeDecimal*".

(La valeur 1733 que désigne *SommeDecimal* est alors recopiée dans le casier *Decimal*, comme en **(1)** dans le schéma qui suit (page 151). Le dialogue continue.)

Notons dès à présent, comme la création du paramètre *Décimal* l'a montré, qu'un paramètre **valeur** est en quelque sorte une variable locale dont la valeur initiale est fixée lors de l'appel par recopiage au sein de ce paramètre d'une valeur précisée par l'appelant.

13 Ce paramètre *Decimal* est de genre valeur simplement parce qu'il **n'est pas** précédé de la mention var.

IE : "Voilà pour le paramètre valeur *Decimal*. Au tour à présent du paramètre *Romain*. Mais, il s'agit cette fois d'un paramètre **variable**, puisque son nom est précédé du mot «var»."

(Il dispose en effet de l'entête `procedure ROMANISER(Decimal: LongInt; var Romain: NombreRomain).` Avec en main l'étiquette *Romain* , il se tourne une nouvelle fois vers l'exécutant principal :)

IE : "J'ai ici une étiquette à coller **en surcharge d'une des tiennes**, sur un de tes casiers. Sur lequel de tes casiers, de type NombreRomain, puis-je la coller ?"

EP : (qui dispose de l'instruction d'appel `ROMANISER(SommeDecimal ,SommeRomain)` "Tu peux coller ton étiquette sur mon casier marqué *SommeRomain*."

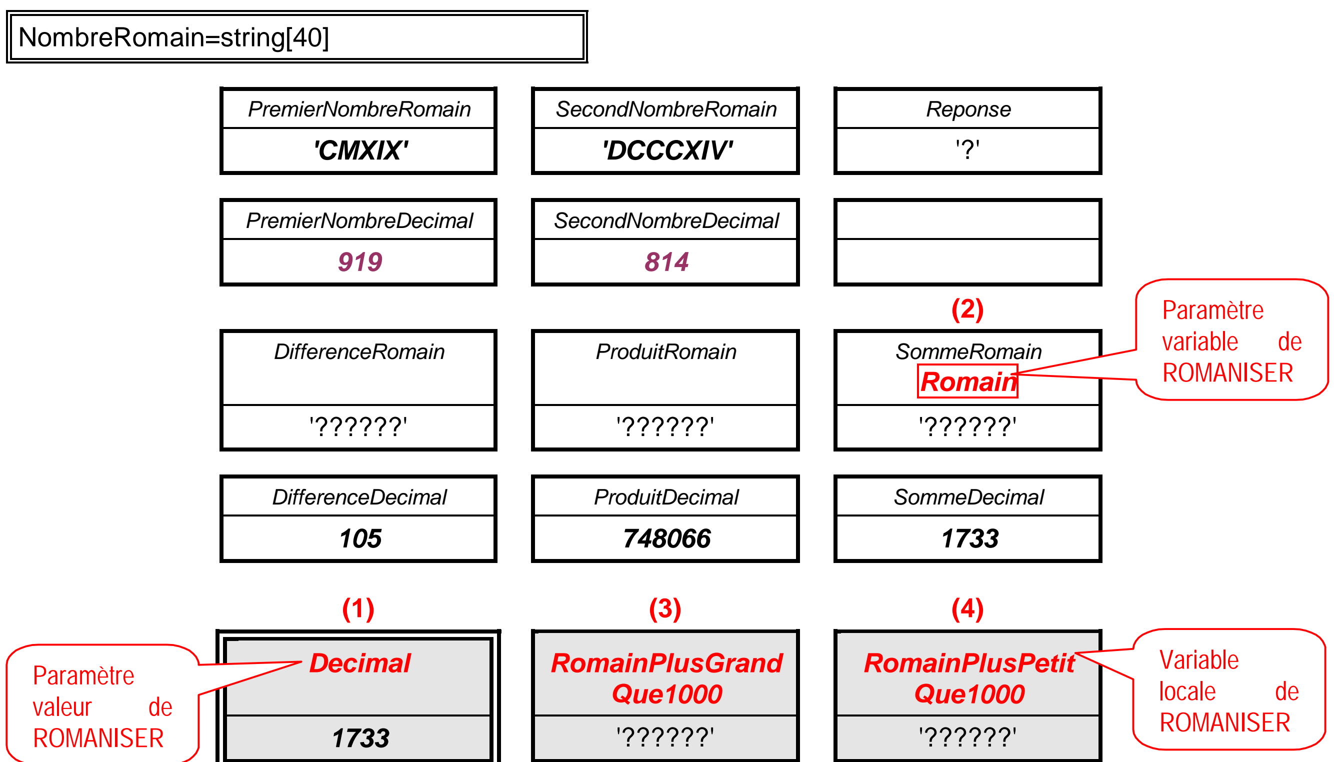
(L'installateur surcharge alors l'étiquette *SommeRomain* avec son étiquette à lui, *Romain*, comme indiqué en **(2)** sur le schéma. Il constate ensuite que la liste de ses paramètre est clôturée. Il se retourne vers l'exécutant :)

IE : "Merci. Je n'ai plus rien à te demander".

(L'exécutant principal se retire du devant de la scène. L'installateur n'en a cependant pas encore terminé.)

IE : "Fini avec les paramètres ! Mais il me reste deux variables locales (*RomainPlusGrandQue1000* et *RomainPlusPetitQue1000*) à installer. Je repère donc 2 casiers vierges, de type NombreRomain -je consulte un moment l'aide mémoire que m'a laissé l'exécutant précisant que le type NombreRomain est string[40]- voilà, *RomainPlusGrandQue1000* et *RomainPlusPetitQue1000*, sont installées."

(Il colle les 2 étiquettes sur des casiers vierges du type adéquat, comme indiqué en **(3)** et **(4)** ci-dessous.)



Les casiers "locaux" à ROMANISER, qu'ils s'agissent de vraies variables locales ou de paramètres valeurs, sont représentés en grisés sur le schéma, les paramètres valeur étant assortis

d'un double encadré. Les étiquettes installées par ROMANISER apparaissent en gras (et rouge si la couleur est disponible). Bien entendu, toutes les variables représentées sont accessibles, y compris la variable qui porte à la fois l'ancien nom *SommeRomain*, mais aussi le nouveau (correspondant au paramètre variable) *Romain*.

En faisant dès à présent le point à propos des paramètres, nous pouvons résumer les différences :

- un paramètre **valeur** est une variable locale, mais qui est remplie à l'appel de la procédure par le programme ou la procédure appelante avec une donnée du type voulu (le type du paramètre);
- un paramètre **variable** (introduit dans la liste des paramètres par le mot «var ») n'est qu'une étiquette qui va venir surcharger une variable existante, du type adéquat; il n'y a donc pas création d'une variable, mais simplement, une variable existant au moment de l'appel est rebaptisée (comme *SommeRomain* qui prend le surnom *Romain*).

1.10.3 Début de l'exécution de ROMANISER et appel de ROMANISER_TOUT_OU_PARTIE

Le local étant celui qui est représenté ci-dessus, l'exécutant auxiliaire chargé de ROMANISER effectue le travail commandé par la partie exécutable de cette procédure (page 141) : constatant que *Decimal* (contenant 1733) est inférieur à 4000, il appelle à son tour un couple auxiliaire chargé de ROMANISER_TOUT_OU_PARTIE :

```
      ROMANISER_TOUT_OU_PARTIE(Decimal,false,Romain);
```

Le couple appelé s'avance; comme l'installateur d'étiquettes correspondant constate l'existence de paramètres

```
procedure ROMANISER_TOUT_OU_PARTIE
(Habituel : LongInt; AuDelaDe1000:boolean; var Romain : NombreRomain);
```

un dialogue va à nouveau s'établir entre lui et l'exécutant de ROMANISER (qui vient de l'appeler). Ce dialogue illustre à nouveau le rôle des paramètres valeurs et variables :

IE : "J'ai un **premier** paramètre **valeur** *Habituel*, de type entier long à installer. Je colle donc l'étiquette *Habituel* sur un casier vierge".

(Se tournant alors vers l'exécutant ROMANISER qui vient de l'appeler :)

IE : "Quelle donnée de type entier long dois-je placer dans mon casier *Habituel* ?"

ER : (exécutant chargé de ROMANISER, consultant l'instruction d'appel `ROMANISER_TOUT_OU_PARTIE(Decimal,false,Romain))` : "Il faut y copier la donnée (contenue dans) *Decimal*."

(L'installateur place donc 1733 dans *Habituel* (comme en **(1)** sur le schéma ci-dessous, page 153). Le dialogue se poursuit.)

IE : "Au tour du **deuxième** paramètre **valeur** *AuDelaDe1000*; j'installe cette étiquette sur un casier vierge de type booléen."

(Se tournant à nouveau vers l'exécutant ROMANISER :)

IE : "Quelle donnée de type booléen dois-je à présent placer dans ce casier *AuDelaDe1000* ?"

ER (consultant l'instruction d'appel `ROMANISER_TOUT_OU_PARTIE (Decimal, false, Romain) )` : "Il faut y placer la constante «faux » («false»)."

(L'exécutant ROMANISER puise alors «faux» dans la malle à information (Cf. Volume 1, page 137) et transmet cette donnée à l'installateur de ROMANISER_TOUT_OU_PARTIE qui la place dans *AuDelaDe1000* (comme en **(2)** sur le schéma ci-dessous). Le dialogue se poursuit.)

IE : "Le **troisième** et dernier paramètre *Romain* à présent. Il s'agit d'un paramètre **variable**, puisqu'il est précédé dans la liste du mot «var»." (Se tournant vers l'exécutant chargé de ROMANISER) : "Sur quelle variable qui était là avant mon arrivée puis-je coller mon étiquette *Romain* ?"

ER (consultant l'instruction d'appel `ROMANISER_TOUT_OU_PARTIE (Decimal, false, Romain)`) : "Il faut placer ton étiquette en surcharge de la variable préexistante qui porte l'étiquette *Romain*."

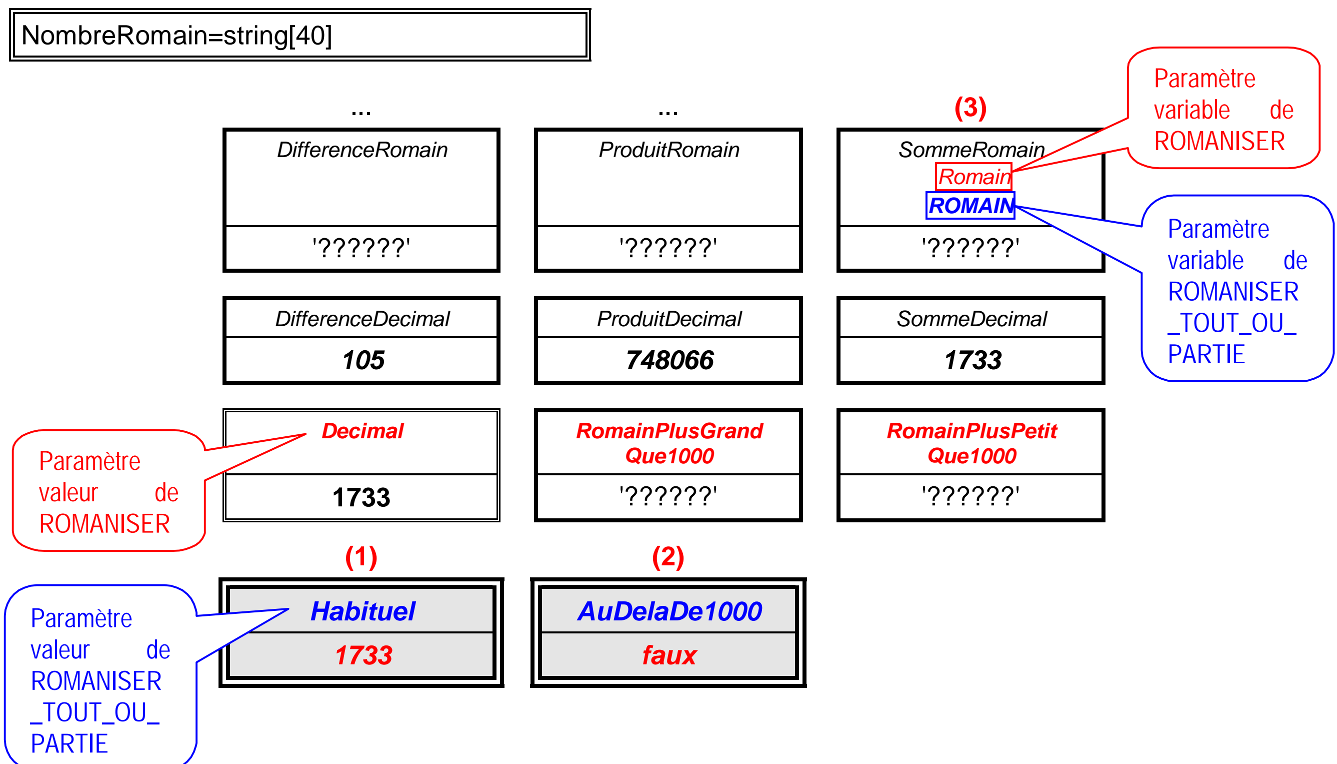
L'installateur place dès lors une troisième étiquette sur le casier qui s'appelle au choix *SommeRomain*, *Romain* (placé par ROMANISER) ou *Romain* (placé par ROMANISER_TOUT_OU_PARTIE). (comme en **(3)**, où l'étiquette la plus récente est figurée en majuscule (et en bleu pour ceux qui ont la couleur))).

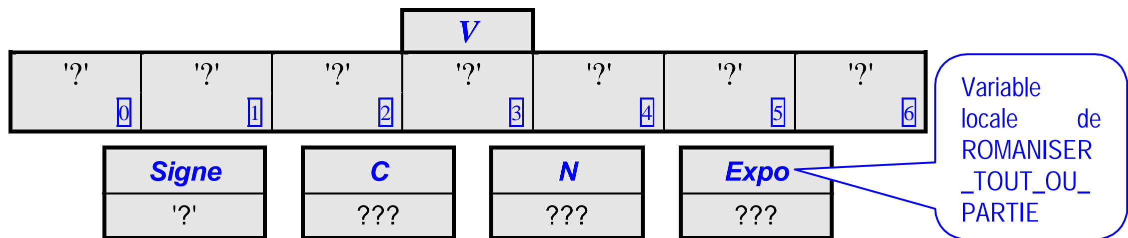
La liste des paramètres de ROMANISER_TOUT_OU_PARTIE étant épuisée, le dialogue prend alors fin et ROMANISER va rejoindre en coulisse l'exécutant principal, laissant le champ libre à l'exécutant chargé de ROMANISER_TOUT_OU_PARTIE.

L'installateur n'a cependant pas terminé puisqu'il lui reste à installer les variables locales :

```
var V : array[0..6] of char;
    Signe : string[1];
    C : -2..6;
    N, Expo :integer;
```

comme sur le bas du schéma ci-dessous (où certaines variables du programme principal n'ont plus été représentées, pour gagner de la place) :



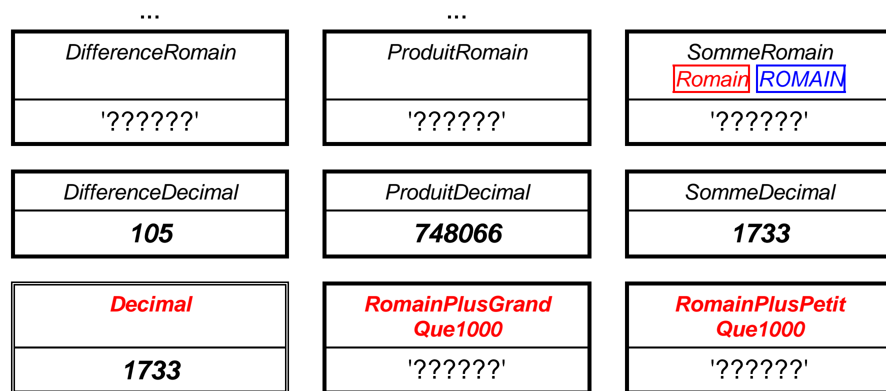


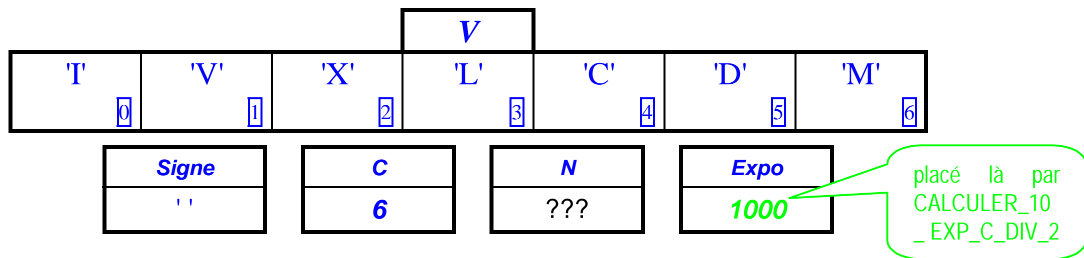
Les paramètres valeurs et les variables locales propres à la procédure ROMANISER_TOUT_OU_PARTIE sont représentés en grisé. Les étiquettes installées par ROMANISER_TOUT_OU_PARTIE sont représentées en gras (et en bleu). (Celles installées par ROMANISER restent en rouge).

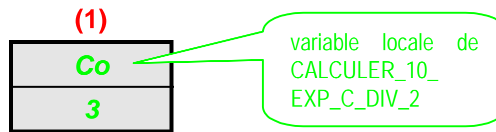
1.10.4 Le travail de ROMANISER_TOUT_OU_PARTIE et l'appel de CALCULER_10_EXP_C_DIV_2

Le travail de l'exécutant chargé de ROMANISER_TOUT_OU_PARTIE peut alors commencer (Cf. page 146). Ainsi, le tableau *V* va être garni, *Romain* et *Signe* également; puis le compteur *C* recevra la valeur 6 et un nouvel exécutant auxiliaire chargé de CALCULER_10_ EXP_C_DIV_2 sera appelé. Comme cette procédure ne possède pas de paramètre aucun dialogue n'a lieu. L'installateur correspondant se contente d'ajouter à la multitude des casiers déjà disponibles un dernier casier local *Co*, comme indiqué en **(1)** ci-dessous.

L'exécutant chargé de CALCULER_10_EXP_C_DIV_2, sur base de *C* (qui contient 6) remplit le casier *Expo* (avec 1000), comme indiqué dans sa marche à suivre (page 147).



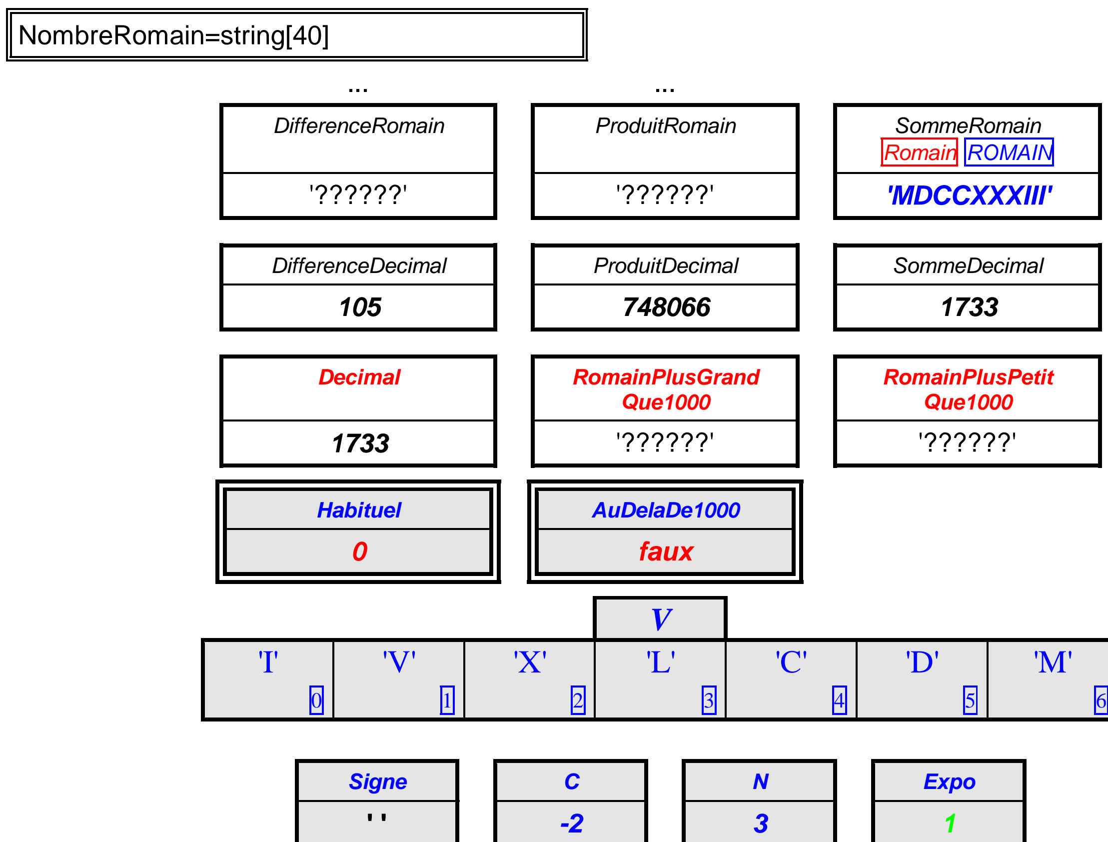




Il termine, son installateur d'étiquette **reprend l'étiquette** *Co* et ils rendent la main à celui qui les avait appelés : ROMANISER_TOUT_OU_PARTIE, laissant un local où le seul élément qui s'est modifié est le contenu du casier *Expo* qui vaut à présent 1000.

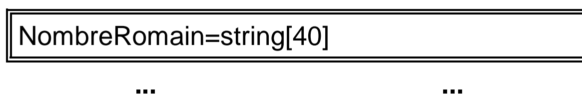
1.10.5 Poursuite et fin du travail de ROMANISER_TOUT_OU_PARTIE

L'exécutant chargé de ROMANISER_TOUT_OU_PARTIE poursuit sa tâche qui comportera encore plusieurs appels de CALCULER_10_ EXP_C_DIV_2. Il finit par terminer avec un local qui ressemble alors à :



A noter le contenu de la variable triplement étiquetée (que ROMANISER_TOUT_ OU_PARTIE appellait lui *ROMAIN*): le travail effectué a conduit à y placer l'équivalent en chiffres romains du contenu de *Decimal*.

ROMANISER_TOUT_OU_PARTIE termine, son installateur d'étiquettes *reprend toutes leurs étiquettes* et ils rendent à ROMANISER (qui les avait appelé) un local :



... ...

DifferenceRomain	ProduitRomain	SommeRomain Romain
'??????'	'??????'	'MDCCXXXIII'

DifferenceDecimal	ProduitDecimal	SommeDecimal
105	748066	1733

Decimal	RomainPlusGrandQue1000	RomainPlusPetitQue1000
1733	'??????'	'??????'

1.10.6 Fin du travail de ROMANISER

Dès que ROMANISER_TOUT_OU_PARTIE lui a rendu la main, ROMANISER, qui en était resté à l'instruction

```
ROMANISER_TOUT_OU_PARTIE(Decimal,false,Romain);
```

termine lui aussi. Son installateur d'étiquettes reprend leurs étiquettes et tous deux rendent la main à l'exécutant principal qui retrouve alors un local :

NombreRomain=string[40]

PremierNombreRomain	SecondNombreRomain	Reponse
'CMXIX'	'DCCCXIV'	'?'

PremierNombreDecimal	SecondNombreDecimal	
919	814	

DifferenceRomain	ProduitRomain	SommeRomain
'??????'	'??????'	'MDCCXXXIII'

DifferenceDecimal	ProduitDecimal	SommeDecimal
105	748066	1733

où le seul changement notable est le contenu du casier *SommeRomain*.

1.10.7 Poursuite du travail de l'exécutant principal et nouvel appel de ROMANISER

L'exécutant principal poursuit alors son travail et tombe immédiatement sur l'instruction :

```
ROMANISER(abs(DifferenceDecimal),DifferenceRomain);
```

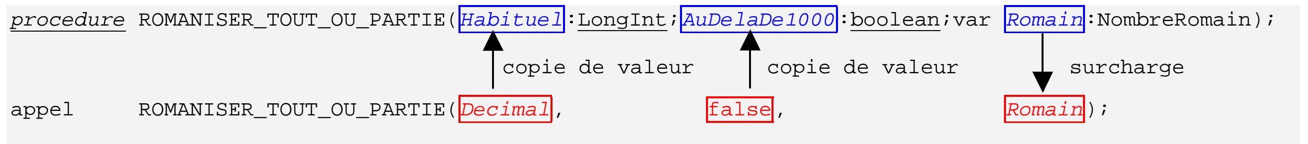
Dès lors, nouvel appel de ROMANISER avec un dialogue établissant la correspondance :

```
procedure ROMANISER(   Decimal:LongInt;     var  Romain:NombreRomain);
                          copie de                   surcharge
                          valeur                     d'étiquette
appel     ROMANISER(abs(DifferenceDecimal), DifferenceRomain);
```

Ainsi, le paramètre valeur (pseudo "variable locale") *Decimal* reçoit la valeur de abs(*DifferenceDecimal*), soit 105 et le casier *DifferenceRomain* est surchargé d'une nouvelle étiquette *Romain*, correspondant au paramètre variable *Romain*.

Comme ROMANISER comporte, en plus des paramètres, une liste de variables locales, ces dernières (*RomainPlusGrandQue1000* et *RomainPlusPetitQue1000* sont installées). Le travail de

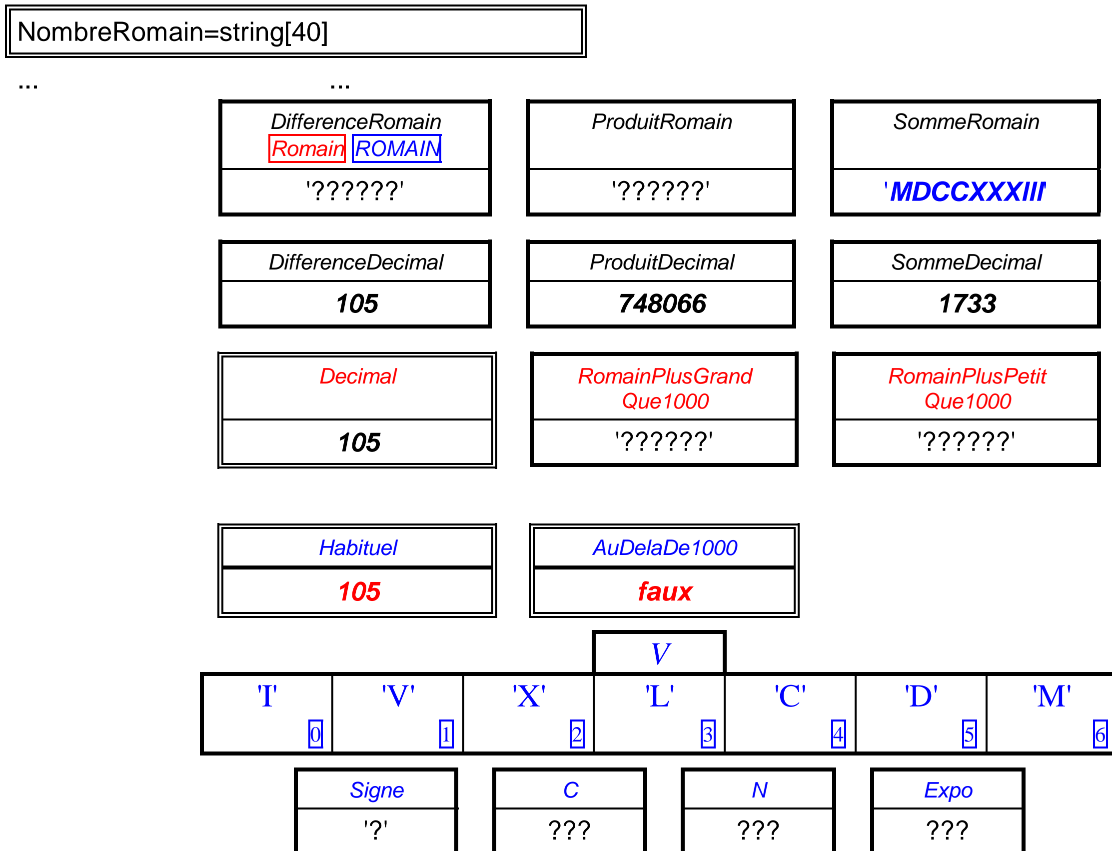
l'exécutant chargé de ROMANISER se déroule et provoque un nouvel appel de ROMANISER_TOUT_OU_PARTIE :



Le dialogue est identique à celui qui a déjà été décrit et conduit donc à l'installation de deux variables étiquetées *Habituel* et *AuDelaDe1000* et qui reçoivent respectivement d'une part la valeur contenue dans *Decimal*, (qui provenait elle-même de *DifferenceDecimal*) soit 105 et d'autre part la constante booléenne «false». Par ailleurs, le paramètre *Romain* étant du genre valeur, l'étiquette *ROMAIN*[14] est à nouveau collée en surcharge de l'étiquette *Romain* (cette dernière surchargeant déjà *DifferenceDecimal*).

Les variables locales *V*, *Signe*, *C*, *N* et *Expo* sont à leur tour installées.

Le local finit donc par ressembler à :

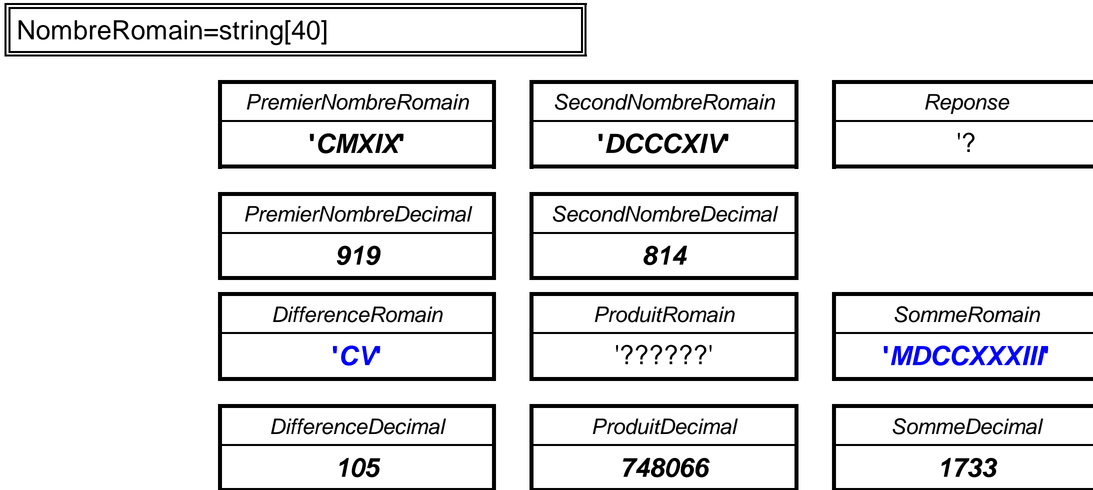


ROMANISER_TOUT_OU_PARTIE débute alors son travail. La procédure CALCULER_10_EXP_C_DIV_2 est à son tour appelée et, pendant un temps, une variable qui lui est propre, *Co*, sera installée.

A l'issue de l'exécution de ROMANISER_TOUT_OU_PARTIE, les variables locales et les paramètres valeurs correspondant disparaissent et l'étiquette *ROMAIN* surchargeant *Romain* également. C'est ensuite ROMANISER qui reprend la main, termine le travail entamé et interrompu

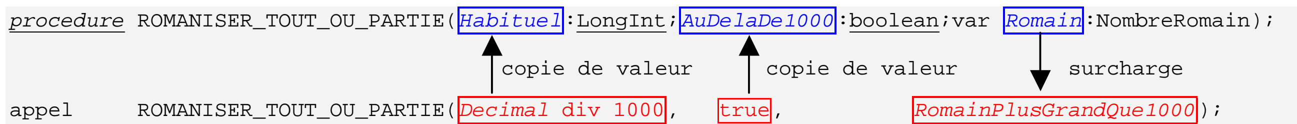
14 Cette étiquette est écrite en majuscules seulement pour la différencier de la précédente, qui porte le même nom. On le sait, en Pascal, l'emploi des majuscules ou minuscules est syntaxiquement non significatif.

par l'appel de ROMANISER_TOUT_OU_PARTIE, en faisant disparaître les étiquettes correspondant aux variables locales, paramètres valeurs et paramètres variables, laissant un local :

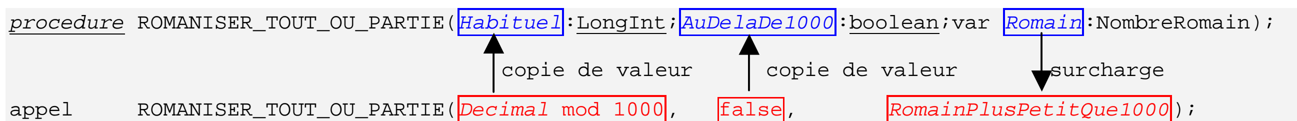


Le travail de l'exécutant principal reprend avec un nouvel appel de ROMANISER :

Cette fois ROMANISER va appeler à deux reprises ROMANISER_TOUT_OU_PARTIE :

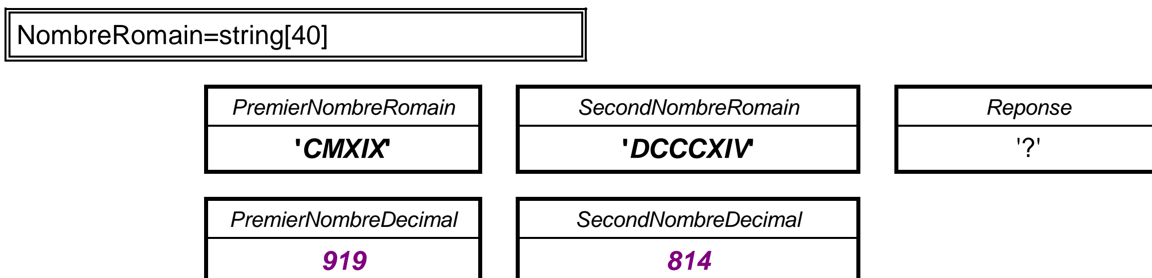


et



Notons donc bien que l'exécutant chargé d'exécuter ROMANISER_TOUT_OU_PARTIE va, à chaque fois qu'il est appelé, effectuer son travail sur base du contenu des deux paramètres valeurs *Habituel* et *AuDelaDe1000*, pour placer le résultat de la transformation dans le paramètre variable *Romain*. Mais à chaque appel, ces deux paramètres valeurs se verront affecter, par l'appelant, des valeurs différentes tandis que le nom du paramètre variable sera à chaque fois un surnom pour une variable préexistante différente choisie par le même appelant.

A l'issue du travail, l'exécutant principal se trouve avec un local :



DifferenceRomain	ProduitRomain	SommeRomain
'CV'	' D°C°C°X°L°V°I°I°I°LXVI'	'MDCCXXXIII'

DifferenceDecimal	ProduitDecimal	SommeDecimal
105	748066	1733

2. Les paramètres d'une procédure

2.1 Pourquoi

Comme l'a montré l'exemple précédent, il arrive que, lors de l'analyse par affinements successifs mise en oeuvre dans la démarche descendante, des actions complexes s'avèrent extrêmement semblables, conduisant dès lors à des procédures jumelles. En réalité, la seule chose qui distingue ces procédures ce sont les variables globales consultées et modifiées.

Dans ce cas plutôt que de recommencer à plusieurs reprises la description de procédures dont les textes ne différent de l'une à l'autre que par les noms des variables mentionnées sur lesquelles les mêmes actions sont commandées, on va se contenter d'un seul texte de procédure, assortis d'un ensemble de paramètres. Le texte de la procédure mentionnera alors, au lieu de jeux de variables différents, ces paramètres. C'est lors de l'appel de procédure que chaque paramètre se verra associer une valeur ou une variable.

2.2 Comment

2.2.1 Les principes

1. Dans l'entête de la procédure, on fera suivra son identifiant (= le nom de la procédure) par la liste des divers paramètres dont on décide de la doter, liste enclose dans des parenthèses. On appelle parfois paramètres **formels** ceux dont les noms sont ainsi indiqués.

   Chaque paramètre est caractérisé par son type (comme le sont les variables) et par son genre : valeur ou variable. Le type de chaque paramètre suit le nom (dont il est séparé par le symbole «: »). Un paramètre de genre variable est de plus (en Pascal) précédé du mot «var».

   Ainsi par exemple :

```
procedure AGIR(A: integer; B: integer; var C: integer; D: integer; var E: real;
var F: real; G: real; H: real; I: integer)
```

   Les paramètres y sont les suivants :

A	de type entier (integer)	de genre valeur
B	de type entier (integer)	de genre valeur
C	de type entier (integer)	de genre variable
D	de type entier (integer)	de genre valeur
E	de type réel (real)	de genre variable
F	de type réel (real)	de genre variable
G	de type réel (real)	de genre valeur
H	de type réel (real)	de genre valeur
I	de type entier (integer)	de genre valeur

   Notons qu'on aurait pu regrouper certain des paramètres (comme on le fait parfois aussi pour les variables) et écrire :

```
procedure AGIR(A, B: integer; var C: integer; D: integer; var E, F: real; G, H:
real; I: integer)
```

L'ordre de succession des paramètres est important, ainsi que la distinction entre le genre **valeur** et le genre **variable**. On notera cependant que le mot «var» peut figurer à plusieurs reprises dans la liste des paramètres.

2. Lors de l'appel de la procédure, chaque paramètre va se voir associer "quelque chose" en fonction de son genre et de son type. Cette association se fait en suivant l'ordre des paramètres formels. Ce qui est ainsi associé aux paramètres formels lors de l'appel, est souvent appelé paramètre **effectif** : une **valeur** pour les paramètres de genre **valeur**, une **variable** pour les paramètres de genre **variable**.

   - Les paramètres valeurs se voient associer une valeur (= une donnée) du type attendu. Comme d'habitude, cette donnée peut être décrite comme une constante, une variable ou encore une expression.

     Lors de l'appel, chaque paramètre valeur donne naissance à une variable locale qui sera initialisée à l'aide de la valeur associée à ce paramètre valeur. Une variable est donc créée et la valeur indiquée lui est affectée.

     A la fin de l'exécution de la procédure, ces variables disparaissent et leurs contenus deviennent donc inaccessibles.

   - Les paramètres variables se voient associer une variable existant avant l'appel et accessible à l'appelant : ce sont des **surnoms**, portés par les variables associées jusqu'à la fin de l'exécution de la procédure. Mais, toutes les modifications apportées lors de l'exécution de la procédure (ou de procédures appelées par cette dernière) à un paramètre variable sont en réalité effectuées sur la variable préexistante associée.

     A la fin de l'exécution de la procédure, le surnom correspondant au paramètre variable disparaît, mais la variable correspondante reste bien entendu disponible, avec un contenu généralement modifié.

? S'il n'existait qu'un seul genre de paramètre, serait-il préférable que ce soit le genre valeur ou le genre variable ? Pourquoi ?

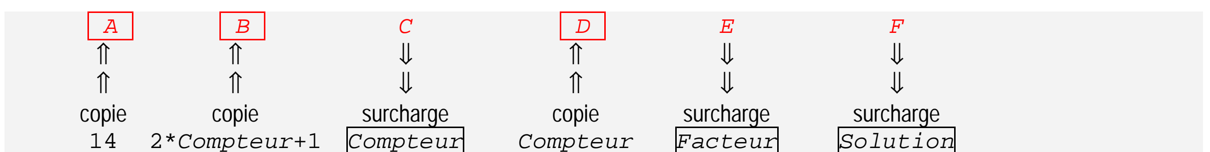
Ainsi, dans le cas de l'exemple donné ci-dessus, on pourrait avoir l' appel suivant :

```
AGIR(14, 2*Compteur+1,Compteur,Compteur, Facteur, Solution, 12.6, Compteur/2, 0)
```

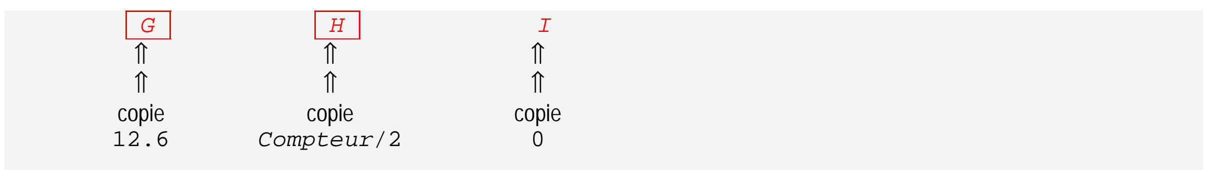
l'entête de la procédure appelée étant :

```
procedure AGIR(A, B: integer; var C: integer; D: integer; var E, F: real; G, H: real; I: integer)
```

la correspondance est alors la suivante :



et



Outre d'autres variables qui auraient été par ailleurs disponibles, on se trouve en tout cas au début de l'exécution de AGIR avec les variables suivantes (en supposant que *Compteur* contienne 15) :

A	*B*	*Compteur* C	*D*
14	**31** (2**Compteur*+1)	**15**	**15** (*Compteur*)

Facteur E	*Solution* F	*G*	*H*
?	**?**	**12.6**	**7.5** (*Compteur*/2)

I
0

3. Comme on le devine, les paramètres valeurs, qui donneront naissance lors de l'appel à des variables locales dans lesquelles seront copiées des données choisies, mais qui disparaîtront à la fin de l'exécution, peuvent servir uniquement à passer de l'information à la procédure appelée; jamais le résultat du travail effectué par une procédure ne pourra être récupéré à travers un paramètre valeur.

   Par contre, comme un paramètre variable n'est qu'un surnom pour désigner, pendant le temps de l'exécution de la procédure, une variable qui existait avant l'appel et continuera à exister après l'exécution (même si le surnom aura alors disparu), un paramètre variable peut servir à recueillir le résultat du travail de la procédure.

   On pourrait dès lors identifier les paramètres valeurs à des manières de fournir en entrée des informations à une procédure et les paramètres variables à des possibilités de sortir des informations à l'issue de l'exécution de la procédure.

? Voyez-vous des situations où il serait judicieux d'utiliser un paramètre variable pour simplement passer en entrée de l'information à une procédure en ne souhaitant pas recueillir quoi que ce soit au travers de ce paramètre à l'issue de l'exécution ? Quelle en est la raison ? Cette manière de procéder comporte-t-elle des risques ?

Identifier paramètre valeur avec entrée d'informations et paramètre variable avec sortie d'informations est généralement correct. Il faut noter cependant qu'un paramètre variable peut évidemment servir aussi à fournir des informations à la procédure appelée et pas seulement à en recueillir. En effet, on peut avoir commandé à l'exécutant principal ou à l'exécutant d'une procédure appelante de ranger des informations indispensables dans une variable qui sera surnommée, lors d'un appel, par un paramètre variable. L'information qui y est ainsi préalablement logée est évidemment accessible, à travers le paramètre variable, par la procédure appelée.

On peut choisir de procéder de cette manière lorsque, par exemple, un tableau de grande dimension doit être passé en entrée à une procédure. Passer les informations contenues dans ce tableau à travers un paramètre valeur provoque la création d'un tableau local à la procédure dans lequel les composantes du tableau original seront recopiées. On se trouve alors, pendant l'exécution de la procédure, avec deux gros tableaux qui vont manger une partie importante des ressources de la mémoire, le tableau local disparaissant seulement à la fin de l'exécution de la procédure. Dans un souci d'épargner la mémoire disponible, on peut choisir d'utiliser un paramètre variable qui surchargera pendant l'exécution de la procédure l'étiquette du tableau original. On n'a plus alors qu'un seul tableau à loger en mémoire pendant l'exécution de la procédure.

Il faut se rendre compte cependant que, dans cette manière de faire, toutes les manipulations effectuées sur le paramètre variable au sein de la procédure le sont en réalité sur le tableau original que ce paramètre désigne. Si donc le tableau original ne pouvait être modifié (mais seulement consulté) par la procédure, il faudra veiller à ce qu'aucun changement ne soit apporté au tableau désigné par le paramètre variable : ce serait l'original qui serait affecté par ces changements. Il s'agit d'une source d'erreur bien connue en programmation : l'**effet de bord**.

4. On parle souvent pour les paramètres valeurs de **passage par valeur** (lors de l'appel) : une variable locale est créée et une **valeur** lui est affectée (= une donnée y est recopiée).

   Dans le cas des paramètres variables, on parle plutôt de **passage par adresse**; c'est non pas une valeur qui est associée au paramètre variable, mais l'**adresse d'une variable** préexistante, autrement dit on précise l'emplacement (en mémoire) de la variable désignée par le paramètre variable.

2.2.2 Les détails syntaxiques en Pascal

2.2.2.1 Nécessité de prédéfinir un type explicite pour les paramètres de type structuré

Comme les variables, les paramètres, tant du genre valeur que du genre variable, peuvent être de l'un quelconque des types déjà rencontrés : entier (integer et les types dérivés dans l'implémentation Turbo Pascal : byte, word, shortint, longint), caractère (char), réel (real et single, double, extended, comp), booléen, énuméré ou intervalle.

Dans le cas de types structurés (comme un tableau, ou d'autres types que vous découvrirez plus tard comme ensemble ou enregistrement), il est cependant indispensable qu'un type explicite ait préalablement été défini. Ainsi, on **ne** peut **pas** écrire

```
procedure TRAITER(T: array[1..100] of real, ...)
```

il est indispensable d'avoir prédéfini, à un endroit du programme que la procédure appelée "connaît" (dans la partie déclaration du programme principal ou d'une procédure "ancêtre) un type explicite :

```
type Etagere = array[1..100] of real;
```

pour pouvoir alors écrire

```
procedure TRAITER(T: Etagere, ...)
```

Cependant, Turbo Pascal admet des paramètres de type string, mais on ne peut en limiter la longueur possible comme dans string[40]; dans ce dernier cas il faut à nouveau avoir prédéfini un type explicite, par exemple :

```
type NombreRomain = string[40]
```

ce qui permettra de définir des paramètres de type NombreRomain[15].

2.2.2.2 Importance de la concordance des types lors des appels

Nous savons que, lors de l'appel, les paramètres effectifs, valeurs (constantes, variables ou expressions) ou variables, associés aux divers paramètres formels sont simplement listés en étant séparés par des virgules, dans l'ordre où les paramètres formels correspondants sont définis dans l'entête de la procédure. Il faut évidemment être attentif à ce que le type de la valeur ou de la variable associées à chaque paramètre formel lors de l'appel coïncide.

[15] Certains de ces détails syntaxiques sont susceptibles de se modifier dans certaines implémentations de Pascal : une fois de plus, si la connaissance du langage utilisé est indispensable, ce n'est pas elle qui constitue l'essentiel.

2.2.2.3 *Les variables surnommées par des paramètres variables restent directement accessibles*

J'ai été attentif dans les schémas proposés à illustrer le fait que les étiquettes correspondant aux paramètres variables, et qui viennent surcharger des étiquettes de variables existantes et accessibles, laissent ces dernières apparentes.

C'est qu'en effet, ces variables restent à la fois accessibles par les surnoms que constituent les paramètres variables associés, mais également directement sous leur nom. Cette possibilité ne constitue cependant pas une pratique recommandable et l'on est en droit de se demander si elle a vraiment du sens au sein d'une saine programmation.

3. Compléments sur Pascal

3.1 *La structure "Case of..."*

Le problème précédent nous a permis d'introduire une nouvelle structure de contrôle "Au cas où" (case... of...) permettant de rendre compte, dans certains cas, d'une série d'altenatives multiples imbriquées.

La structure générale de l'instruction "*case*... *of*..." est la suivante :

```
case expression de sélection (1) of

   ensemble de valeurs constantes
   possibles de l'expression de sélection(2) : instruction unique à effectuer dans ce cas; (3)

   ensemble de valeurs constantes
   possibles de l'expression de sélection   : instruction unique à effectuer dans ce cas;

...       ...              :...       ...
else (4)
                                            instruction unique à effectuer dans ce cas;
end (5);
```

On notera les points suivants :

(1) L'**expression** dont les diverses valeurs possibles vont entraîner le choix de l'instruction à exécuter doit être de type **scalaire** : entier (ceci recouvrant les divers types d'entiers manipulés : integer, longint, byte,...), caractère, booléen, énuméré et intervalle.

Cette expression peut bien entendu se réduire à un nom de variable d'un de ces types mais ne peut évidemment pas être une constante.

On notera que l'expression de sélection **ne peut être** de type réel, ni de type chaîne de caractères (string).

Par exemple, on peut écrire :

```
case Compteur mod 1000 of ....
                                   où Compteur est une variable entière
```

ou

```
case Indice of ....
                                   où Indice est de type caractère
```

mais pas

```
case Compteur / 1000 of ....
                                   où Compteur est une variable entière
```

car l'expression de sélection est alors de type réel

ni non plus

```
case NomDuClient of ....
                             où NomDuClient est une variable de type chaîne (string)
```

(2) Les différents cas envisagés sont présentés en donnant des valeurs possibles de l'expression de sélection; ces valeurs doivent être écrites comme des **constantes** ou des **intervalles bornés par des constantes**, du même type que celui de l'expression de sélection.

Elles ne peuvent en aucun cas être écrites sous le forme de variables ou d'expressions.

Par exemple :

```
case Compteur mod 1000 of
           1, 20, 100..200, 999 : ....
           0, 2..19, 21..99, 101 : ....
                                                                              ... ...
```

ou

```
case Indice of
           'a'..'z' : ....
           'A'..'Z' : ....
                                                                              ... ...
```

mais pas

```
case Compteur mod 1000 of
           Compteur : ....
```

car *Compteur* n'est pas une constante

Les constantes écrites peuvent évidemment avoir été définies par l'instruction const.

Il est syntaxiquement possible qu'une même constante se retrouve dans les divers cas envisagés. Il faut cependant savoir que ces divers cas sont examinés l'un après l'autre, dans l'ordre où ils se présentent, et que, dès que la valeur de l'expression de sélection est repérée, l'instruction correspondante est exécutée et les cas suivants sont ignorés. Autrement dit, la structure "*case*... *of*..." reproduit une cascade de "*if*... *then*... *else*..." imbriqués et non une simple suite de "*if*... *then* ..." .

(3) Derrière chaque liste de constantes constituant l'un des cas figure l'instruction unique à exécuter dans ce cas; comme d'habitude, si plusieurs instructions sont à exécuter, elles seront encloses entre les mots "begin" et "end" pour donner l'impression que leur ensemble constitue une seule instruction. (Cf. volume 1, page 221).

On notera que la liste des constantes est séparée de l'instruction à exécuter par le symbole «**:**».

(4) Il peut arriver qu'aucune des constantes listées dans les différents cas envisagés ne corresponde à la valeur actuelle de l'expression de sélection. Dans ce cas, si une mention "*else*" figure dans la structure, c'est l'instruction suivant ce "*else*" qui sera effectuée. En l'absence de la mention "*else*" (qui est, de fait, facultative) aucune des instructions recensées dans le "*case*... *of*... " n'est effectuée et on passe à l'instruction suivant le "*end*" de ce "*case*... *of*... "

On notera qu'aucun symbole «:» ne sépare le "*else*" de l'instruction à effectuer dans ce cas. On notera également qu'ici le *else* est précédé du symbole «**;**» terminant l'instruction précédente.

(5) La mention "*end*" doit terminer l'écriture du "*case*... *of*... ". C'est la première fois que nous trouvons un "end" qui ne soit pas le pendant d'un "begin".

 © CeFIS - FUNDP 2002

4. Des chiffres et des lettres

Nous allons à présent aborder un exercice difficile, mais qui nous permettra de cerner davantage encore les limites de l'approche descendante. Jusqu'ici, même lorsqu'elles étaient assorties de paramètres (afin de permettre plusieurs appels successifs) les procédures n'étaient jamais appelées que par leur procédure-mère (ou leur programme-père).

Nous allons maintenant montrer que les possibilités d'appel entre procédures débordent (largement) la simple généalogie : une procédure (ou le programme principal) pourra se permettre d'appeler d'autres procédures que ses "filles".

Cette constatation est fondamentale et mettra un terme, **dans certains cas**, à une approche trop strictement descendante.

Il ne faudrait pas cependant croire que l'approche descendante prônée dans le chapitre précédent et qui nous a tenu lieu de méthode doit ou peut être oubliée : elle restera notre guide, même si elle ne doit pas être interprétée trop strictement.

Enfin, le problème abordé ici est particulièrement complexe. Il va donc falloir s'accrocher et, surtout, garder sous les yeux, aussi souvent que possible, la totalité de la structure de la solution proposée.

4.1 Les nombres en toutes lettres : description floue

La tâche concernée consiste tout bonnement à écrire "en toutes lettres" un nombre fourni "en "chiffres". Par exemple "312" doit devenir "trois cent douze".

Il serait intéressant, qu'avant de poursuivre, le lecteur passe directement à l'annexe page 207, traitant des chaînes de caractères, pour découvrir quelques-uns des outils qui seront utilisés ici. Il sera toujours temps ensuite de revenir au présent problème.

4.2 Les nombres en toutes lettres : "Quoi faire ?"

Avant d'en venir à l'habituelle série de questions qui vont permettre de préciser le propos, voici, en guise d'introduction quelques écrans qui présentent le type de traitement attendu :

```
Vous allez me fournir un nombre en chiffres et je l'écrirai
en toutes lettres!

Les nombres peuvent comporter une partie décimale d'au plus trois
chiffres et ne peuvent comporter plus de six chiffres dans leur partie
entière.
Frappez Entrée pour poursuivre
```

```
Donnez le nombre à écrire : 421

Ce nombre s'écrit :

quatre cent vingt et un


   On reprend (O ou N) ?O
```

```
Donnez le nombre à écrire : 880880.800

Ce nombre s'écrit :

huit cent quatre-vingt mille huit cent quatre-vingts et huit dixièmes


   On reprend (O ou N) ?O
```

```
Donnez le nombre à écrire : 999999.999

Ce nombre s'écrit :

neuf cent nonante-neuf mille neuf cent nonante-neuf et neuf cent
nonante-neuf millièmes


   On reprend (O ou N) ?O
```

```
Donnez le nombre à écrire : 0.450

Ce nombre s'écrit :

quarante-cinq centièmes


   On reprend (O ou N) ?O
```

```
Donnez le nombre à écrire : 560007.55555
```

```
Donnez le nombre à écrire : 0.0

Ce nombre s'écrit :

 zéro

   On reprend (O ou N) ?N
```

On le voit, il s'agit essentiellement d'écrire un nombre "en toutes lettres". Nous référant au tableau de la page 120, qui organisait les questions à poser lors de cette importante étape du "Quoi faire ?", nous pouvons apporter les précisions suivantes :

	Monde de la tâche	Monde de l'exécutant
ENTREES	*Taille des nombres à écrire ?* Au plus 6 chiffres pour la partie entière et 3 pour la partie décimale *Reprise du traitement ?* Oui, à la demande de l'utilisateur.	*Quel peut être le séparateur entre la partie entière et la partie décimale (point ou virgule) ?* Au choix; ce peut donc être un point. Mais ce serait bien qu'on puisse indifféremment écrire un point ou une virgule *Faut-il toujours une partie décimale ? et une partie entière ?* C'est comme on veut. Ce serait bien de pouvoir écrire 57, 57.0, 0.60, .60,...
SORTIES	*Ecriture "à la belge" (septante...)?* Oui. *Comment qualifier la partie décimale ?* Oublier les éventuels 0 terminaux et écrire "dixième", "centième" ou "millième" (éventuellement au pluriel) *Des majuscules ou des minuscules ?* Des minuscules.	Cf. les écrans proposés ci-dessus. *Combien d'espaces entre les mots constituant l'écriture ? (un seul ?)* C'est sans importance, pour autant qu'on ne les multiplie pas trop.
TRAITEMENTS		*Que faire si les nombres fournis n'ont pas les caractéristiques voulues ?* On accepterait volontiers qu'aucune vérification et aucune validation des données ne soit faite. Mais ce serait mieux si les nombres ne répondant pas au format exigé étaient simplement ignorés, une nouvelle donnée étant alors demandée.

Il y aurait sans doute d'autres questions à poser. C'est malheureusement, qu'on le veuille ou non, lorsqu'on est engagé dans l'analyse, qu'un certain nombre de questions supplémentaires (essentiellement celles émanant du monde de l'exécutant) vont apparaître. Avec les précisions apportées ci-dessus, on voit cependant assez clairement ce qu'il reste à faire faire...

4.3 Les nombres en toutes lettres : "Comment faire à un premier niveau ?"

La tâche évoquée va exiger que nous nous remettions en mémoire quelques unes des règles qui président à l'écriture des nombres en toutes lettres. Il nous faut donc rouvrir notre "Grévisse" au chapitre traitant des "adjectifs numéraux cardinaux" ([Grévisse 80], page 437 et suivantes).

Nous y apprenons entre autre que :

859 "Dans les adjectifs numéraux composés, on met le trait d'union entre les éléments qui sont l'un et l'autre moindres que cent, sauf s'ils sont joints par **et**, qui remplace alors le trait d'union."

860 "La conjonction **et** ne s'emploie dans les noms de nombre que pour joindre un aux dizaines (sauf **quatre-vingt-un**)."

866 "En général, les adjectifs numéraux cardinaux, même employés substantivement, sont invariables. Seuls **un**, **vingt**, **cent** peuvent varier. Il faut joindre à ces mots **mille**, qui fait l'objet de certaines remarques."

873 "**Vingt** et **cent** prennent un s quand ils sont multipliés par un autre nombre et qu'ils terminent l'adjectif numéral."

876 "**Mille**, adjectif numéral, est toujours invariable".

Le problème du pluriel pour **vingt** et **cent**, lorsque le nombre est suivi d'une partie décimale, se règle comme si cette partie décimale était absente : on écrit **deux cent mille deux cents** et **deux cent mille deux cents et huit dixièmes**.

Une fois de plus, la connaissance de ces règles rend la **tâche** d'écriture des nombres fort routinière. C'est évidemment un tout autre **problème** que de les **faire** écrire...

Il semble dès à présent évident qu'il sera nécessaire de disposer de la partie entière (celle précédant la virgule ou le point) et de la partie décimale du nombre à écrire. La stratégie globale d'écriture s'exprime alors dans le tableau de choix suivant :

	Partie décimale nulle	*Partie décimale non nulle*
Partie entière nulle	on écrit "zéro"	on n'écrit que la partie décimale
Partie entière non nulle	on n'écrit que la partie entière	on écrit la partie entière, puis "et " puis la partie décimale

Tenant compte de ce tableau, à un tout premier niveau d'analyse, les étapes essentielles pourraient donc par exemple être les suivantes :

- Sur base du nombre reçu, isoler la partie entière et la partie décimale
- **Si** la partie entière est différente de 0 **(1)**
  - Écrire la partie entière
  - **Si** la partie décimale est différente de 0 ajouter "et " **(2)**
- **Si** la partie décimale est différente de 0
  - Oublier les 0 éventuels terminant la partie décimale
  - Ajouter à ce qu'on a déjà écrit l'écriture de la partie décimale ainsi raccourcie
  - Faire suivre la partie décimale de l'indication de sa nature (dixième(s), centième(s), millième(s))
- **Si** la partie entière et la partie décimale sont toutes deux nulles, écrire "zéro" **(3)**

Le tout est à reprendre à la demande

Ce premier jet demande sans doute quelques commentaires :

(1) On n'écrit pas "zéro et trois dixièmes" mais "trois dixièmes"; on ne mentionne donc la partie entière que lorsque celle-ci est non nulle.

(2) Pour devoir écrire "et ", il faut que la partie entière soit non nulle **et** la partie décimale également.

(3) Lorsque la partie entière et la partie décimale sont nulles, on est passé outre des deux "**Si**" qui précédaient. Il faut donc dans ce cas écrire "zéro" car on ne peut pas se contenter de ne rien écrire; mais il ne faut pas, si on a déjà constaté que soit la partie entière, soit la partie décimale est non nulle, passer par cette écriture de "zéro".

? Voyez-vous d'autres manières de rendre compte, au niveau de la stratégie, des alternatives présentes dans le tableau proposé ?

4.4 Les nombres en toutes lettres : "Comment faire faire à un premier niveau ?"

Il nous faut à présent traduire cette stratégie globale en une marche à suivre.

4.4.1 Structure de données

Nous commençons, comme d'habitude, par évoquer l'existence éventuelle de constantes inhérentes au problème posé. On serait évidemment prêt à en avancer deux : le nombre maximal (6) de chiffres de la partie entière et celui (3) de la partie décimale des nombres à traiter.

On devine cependant aisément, que même si on décidait de mettre en évidence la longueur maximale de la partie entière en définissant par exemple, LongueurPartieEntiere = 6, il ne suffirait pas de modifier cette définition en LongueurPartieEntiere = 7, pour que, sans plus d'analyse, le programme à écrire traite les nombres au delà du million : c'est la structure entière du programme qui serait à revoir. Ce serait donner un sentiment factice de facilité à propos des modifications possibles des spécifications que de faire figurer une constante (dont la valeur est très aisément modifiable) dans le programme : être capable de traiter les nombres de 7 chiffres est bien autre chose que de faire passer une constante de 6 à 7...

Rien n'empêche cependant de définir ces 2 constantes, mais, d'une part, je préfère ici garder 6 et 3 pour les désigner, et d'autre part, ce serait peut-être lors de l'analyse de la procédure de lecture et de validation du nombre à traiter que ces constantes pourraient être définies (Voir l'exercice proposé page 207).

Quant aux tableaux qui seraient nécessaires pour le traitement, aucun ne s'impose à ce niveau d'analyse.

Il est par contre une décision, lourde de conséquences, qu'il va nous falloir prendre d'emblée : comment allons-nous représenter la partie entière et la partie décimale du nombre à traiter ? Entiers ou chaînes de caractères ? On le devine dès à présent, les manipulations sur ces parties entières et décimales consisteront essentiellement à les découper (par exemple pour obtenir la partie multiple de 1000 et celle inférieure à 1000), puis à associer aux "nombres" mis en évidence des mots (15 → "quinze", 16 → "seize",...). Dès maintenant, on peut pronostiquer qu'un usage intensif sera fait de la structure "case... of...". Et c'est sans doute cet argument qui est déterminant : l'expression de sélection présente dans la structure case...of...peut être de type entier mais pas de type chaîne de caractères. On pourra donc écrire :

```
case NombreAEcrire of
       11 : write('onze ');
       12 : write('douze ');
       ...
```

NombreAEcrire étant bien entendu de type entier, **mais pas**

```
case NombreAEcrire of
       '11' : write('onze ');
       '12' : write('douze ');
       ...
```

avec *NombreAEcrire* de type chaîne de caractères (Cf. page 163).

Nous décidons dès lors que tant la partie entière que la partie décimale du nombre à traiter seront de type entier. Nous sommes ainsi en mesure d'écrire à un tout premier niveau la marche à suivre souhaitée.

4.4.2 Marche à suivre

```
AVERTIR
Répéter
    LIRE_ET_EVALUER                                   PartieEntiere, PartieDecimale de type entier long
    Affiche : 'Ce nombre s'écrit : ' et passe  quelques lignes
    Si PartieEntiere ≠ 0 alors
        ECRIRE_PARTIE_ENTIERE
        Si PartieDecimale ≠ 0 alors
            Affiche 'et ' (1)
    Si PartieDecimale ≠ 0 alors
        RACCOURCIR                                    PartieRaccourcie de type entier long
        ECRIRE_PARTIE_RACCOURCIE
        FAIRE_SUIVRE_DE_SA_NATURE
    Si PartieEntiere = 0 et PartieDecimale = 0 alors
        Affiche 'zéro ''
    Demande s'il faut reprendre et place la réponse dans Reponse      Reponse de type caractère
jusqu'à ce que Reponse = 'N'
```

L'affichage demandé en **(1)** (comme tous les affichages qui suivront) se fera sans passage à la ligne; on notera de plus que la chaîne 'et ' à afficher finit par un espace, ceci afin de scinder les diverses parties du nombre en toutes lettres à afficher. On va le voir ci-dessous, c'est la même politique qui sera tenue pour tous les affichages commandés : les mots à afficher seront suivis d'un espace et tous les affichages demandés dans l'écriture du nombre se feront sans passage à la ligne (pour avoir une écriture d'un seul tenant).

Comme on le constate, outre *Reponse*, trois variables essentielles sont définies à ce niveau : *PartieEntiere* qui contiendra le nombre entier d'au plus 6 chiffres constituant la partie entière du nombre à traiter (et qui, vu sa taille, doit être de type entier long), *PartieDecimale* qui accueillera la partie décimale et *PartieRaccourcie* qui contiendra le nombre correspondant à la partie décimale privée de ses éventuels zéros terminaux.

Bien que le type entier normal suffise pour *PartieDecimale* et pour *PartieRaccourcie*, je les ai, comme *PartieEntiere,* définis comme étant des entiers longs, pour de simple raisons d'homogénéité.

Bien entendu, comme d'habitude, la marche à suivre proposée n'est réellement compréhensible qu'après précision des spécifications de chacune des actions complexes LIRE_ET_EVALUER, ECRIRE_PARTIE_ENTIERE, ECRIRE_PARTIE_RACCOURCIE, etc..

Quelques explications préalables sont pourtant probablement nécessaires pour saisir le rôle exact de LIRE_ET_EVALUER, de RACCOURCIR et de FAIRE_SUIVRE_DE_SA_NATURE, essentiellement en ce qui concerne la partie décimale du nombre à écrire :

- Comme indiqué ci-dessus, on souhaite une écriture qui, en ce qui concerne la partie décimale se comporte de la manière suivante :

 © CeFIS - FUNDP 2002

Nombre fourni par l'utilisateur	*PartieDecimale* (à la sortie de LIRE_ET_EVALUER)	*PartieRaccourcie* (à la sortie de RACCOURCIR)	Partie décimale écrite par ECRIRE	Texte ajouté par FAIRE_SUIVRE_DE_SA_NATURE
XXX.121	121	121	cent vingt et un	millièmes
XXX.12	120	12	douze	centièmes
XXX.02	20	2	deux	centièmes
XXX.20	200	2	deux	dixièmes

- Dès lors, LIRE_ET_EVALUER fournira toujours dans l'entier *PartieDecimale* **le nombre de millièmes de la partie décimale** du nombre donné par l'utilisateur, même si cette partie décimale ne comportait primitivement que 1 ou 2 chiffres décimaux.
- RACCOURCIR fournira dans *PartieRaccourcie* un entier qui reprend l'entier présent dans *PartieDecimale* privé de ses éventuels zéros terminaux. Ceci est indispensable puisque si *PartieDecimale* comporte par exemple 20 (millièmes), c'est **deux** centièmes qu'il faudra écrire. Il faut donc que sur base de 20 on retrouve 2.
- FAIRE_SUIVRE_DE_SA_NATURE fera suivre de sa nature le nombre présent dans *PartieRaccourcie* et affiché : ainsi, si *PartieDecimale* contenait 20, *PartieRaccourcie* contiendra 2, ECRIRE fera afficher "deux" et c'est "centièmes" qui sera la nature de cette partie affichée.

Il ne faut pas oublier que même si l'utilisateur, fournit le nombre xxx.02, c'est 20 qui est attendu dans *PartieDecimale* et comme cette variable est de type entier (long), il n'y a aucun moyen de distinguer 20 de 020 (ce qui aurait pu être le cas si on avait choisi une représentation sous la forme chaîne de caractères).

Pas de problème dès lors pour la définition de

AVERTIR
Effacer puis afficher un écran d'avertissement de l'utilisateur du type de celui proposé page 165. Attendre jusqu'à la frappe sur Entrée et effacer.

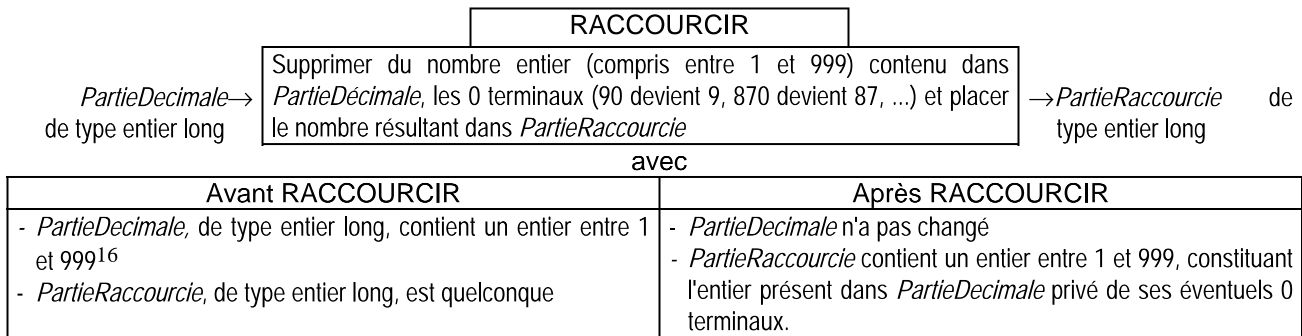
ni non plus pour

LIRE_ET_EVALUER	
Lire un nombre et le refuser (en recommençant une lecture) si son écriture n'est pas celle d'un réel comportant au plus 6 chiffres dans sa partie entière et au plus 3 chiffres dans sa partie décimale. Le séparateur décimal est le point (mais ce serait un plus d'accepter aussi la virgule). On acceptera, dans le cas d'une partie entière nulle, aussi bien l'écriture 0.XX que .XX. Dans le cas d'une partie décimale nulle, on acceptera des écritures comme XX, XX.0, XX.00 ou même XX. La partie décimale du nombre (placée dans *PartieDecimale*) exprimera toujours le nombre de millièmes. La partie entière sera placée dans *PartieEntiere* et le nombre de millièmes constituant la partie décimale dans *PartieDecimale*	→*PartieEntiere*, →*PartieDecimale* de type entier long

avec

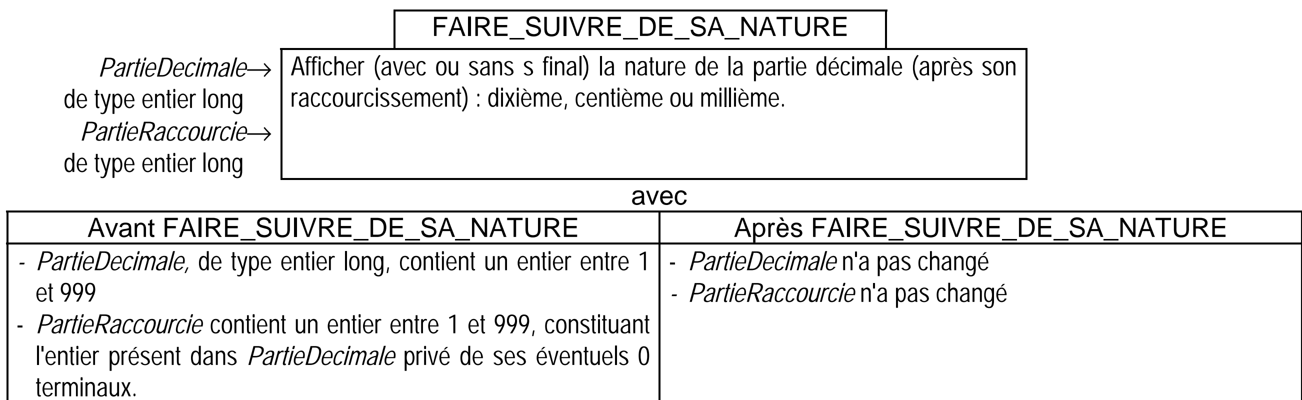
Avant LIRE_ET_EVALUER	Après LIRE_ET_EVALUER
- *PartieEntiere*, de type entier long, est quelconque - *PartieDecimale*, de type entier long, est quelconque	- *PartieEntiere* contient un entier entre 0 et 999999 représentant la partie entière du nombre lu et accepté - *PartieDecimale* contient un entier entre 0 et 999 représentant la partie décimale, exprimée en millièmes, du nombre lu et accepté

ni pour



Avant RACCOURCIR	Après RACCOURCIR
- *PartieDecimale,* de type entier long, contient un entier entre 1 et 999[16] - *PartieRaccourcie*, de type entier long, est quelconque	- *PartieDecimale* n'a pas changé - *PartieRaccourcie* contient un entier entre 1 et 999, constituant l'entier présent dans *PartieDecimale* privé de ses éventuels 0 terminaux.

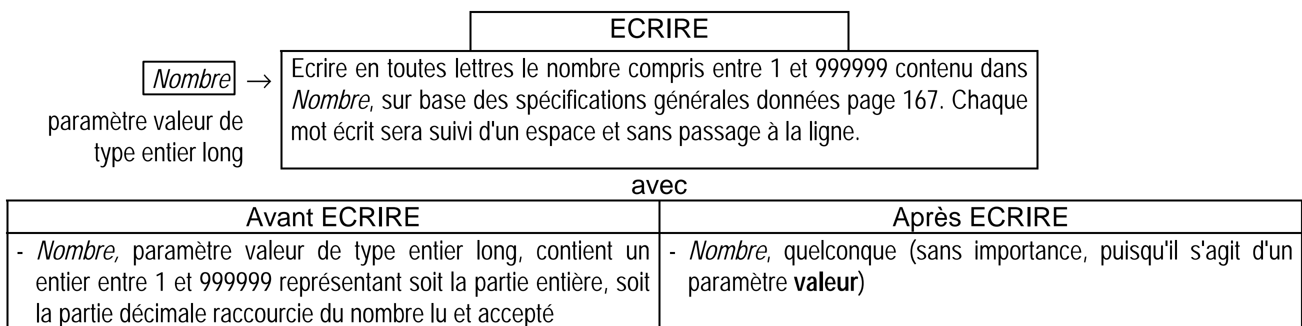
ni pour



Avant FAIRE_SUIVRE_DE_SA_NATURE	Après FAIRE_SUIVRE_DE_SA_NATURE
- *PartieDecimale,* de type entier long, contient un entier entre 1 et 999 - *PartieRaccourcie* contient un entier entre 1 et 999, constituant l'entier présent dans *PartieDecimale* privé de ses éventuels 0 terminaux.	- *PartieDecimale* n'a pas changé - *PartieRaccourcie* n'a pas changé

Par contre, on peut prévoir que ECRIRE_PARTIE_ENTIERE et ECRIRE_ PARTIE_RACCOURCIE vont une fois de plus être très proches l'une de l'autre, puisqu'il s'agira à chaque fois d'écrire en toutes lettres le nombre compris dans *PartieEntiere* pour l'une, *PartieRaccourcie* pour l'autre. Même si la taille de ces nombres est différente, on va pouvoir une fois encore uniformiser ces deux actions en une seule, ECRIRE, assortie d'un paramètre qui accueillera *PartieEntiere* d'une part, *PartieRaccourcie* d'autre part.

On aura donc :



Avant ECRIRE	Après ECRIRE
- *Nombre,* paramètre valeur de type entier long, contient un entier entre 1 et 999999 représentant soit la partie entière, soit la partie décimale raccourcie du nombre lu et accepté	- *Nombre*, quelconque (sans importance, puisqu'il s'agit d'un paramètre **valeur**)

Avec cette définition, la marche à suivre de premier niveau peut se réécrire :

AVERTIR

Répéter

    LIRE_ET_EVALUER   *PartieEntiere, PartieDecimale* de type entier long

    Affiche : 'Ce nombre s'écrit : ' et passe quelques lignes

    Si *PartieEntiere* ≠ 0 alors

---

16 Lorsque RACCOURCIR est appelé, on est certain que *PartieDecimale* n'est pas nulle.

ECRIRE(*PartieEntiere*)
Si *PartieDecimale* ≠ 0 alors
Affiche 'et '
Si *PartieDecimale* ≠ 0 alors
RACCOURCIR — *PartieRaccourcie* de type entier long
ECRIRE(*PartieRaccourcie*)
FAIRE_SUIVRE_DE_SA_NATURE
Si *PartieEntiere* = 0 et *PartieDecimale* = 0 alors
Affiche 'zéro ''
Demande s'il faut reprendre et place la réponse dans *Reponse* — *Reponse* de type caractère
jusqu'à ce que *Reponse* = 'N'

Et nous voici en mesure, dès à présent d'écrire le programme principal correspondant :

4.5 Les nombres en toutes lettres : "Comment dire à un premier niveau ?"

```
program NOMBRE_EN_TOUTES_LETTRES;

       (* ce programme fait lire un nombre et procède à son écriture en toutes lettres Les nombres
       peuvent comporter une partie décimale d'au plus trois chiffres et ne peuvent comporter plus
       de 6 chiffres dans leur partie entière *)
uses WinCrt;

var PartieEntiere,
       (* qui contiendra la partie entière du nombre à traiter *)
    PartieDecimale,
       (* qui contiendra la partie décimale exprimée en millièmes *)
    PartieRaccourcie
       (* qui contiendra le nombre entier constituant la partie décimale privée de ses éventuels
       zéros terminaux *)
    :longint;
    Reponse : char;
       (* pour savoir si on reprend *)

procedure AVERTIR;

procedure LIRE_ET_EVALUER;

       (* elle fait lire un nombre et vérifie qu'il a bien les caractéristiques voulues ( au plus 6
       chiffres pour la partie entière et 3 chiffres pour la partie décimale). Elle fait placer la
       partie entière dans PartieEntiere et la la partie décimale (exprimée en millièmes)dans
       PartieDecimale. Les erreurs éventuelles sont détectées et la lecture est alors recommencée *)

procedure RACCOURCIR;

       (* Elle "raccourcit" éventuellement PartieDecimale en en supprimant les 0 terminaux et place
       le résultat dans PartieRaccourcie *)

procedure FAIRE_SUIVRE_DE_SA_NATURE;

       (* Elle fait suivre la partie décimale telle qu'elle figure (après son raccourcissement) dans
       PartieRaccourcie de sa nature : dixième(s), centième(s) ou millième(s) *)

procedure ECRIRE(Nombre:longint);

       (* Elle fait écrire en toutes lettres Nombre (de type entier long et compris entre 1 et
       999999) Nombre vaudra successivement lors des appels PartieEntiere puis PartieRaccourcie *)

begin
AVERTIR;
repeat
  Clrscr;
  LIRE_ET_EVALUER;
  writeln;
```

```
  writeln('Ce nombre s''écrit : ');
  writeln;
  if PartieEntiere <> 0  then
    begin
    ECRIRE(PartieEntiere);
    if PartieDecimale <> 0 then
      write('et ');          (1)
    end;
  if PartieDecimale <> 0 then
    begin
    RACCOURCIR;
    ECRIRE(PartieRaccourcie);
    FAIRE_SUIVRE_DE_SA_NATURE;
    end;
  if (PartieEntiere = 0) and (PartieDecimale = 0) then
    write(' zéro ');
  Gotoxy(4,20);
  write('On reprend (O ou N) ?');
  readln(Reponse);
  Reponse:=upcase(Reponse);
until Reponse='N';
end.
```

En **(1)**, on notera, comme prévu, l'affichage par write qui ne provoque pas de passage à la ligne et l'espace suivant le mot "et".

Il ne reste plus qu'à poursuivre avec l'analyse de deuxième niveau de chacune des actions complexes spécifiées ci-dessus, en commençant par les plus simples et en laissant pour la bonne bouche ECRIRE et LIRE_ET_EVALUER.

4.6 Analyse de AVERTIR

Je me contenterai de fournir le texte de la procédure correspondante.

```
procedure AVERTIR;

begin
  Clrscr;
  writeln('Vous allez me fournir un nombre en chiffres) et je l''écrirai');
  writeln('en toutes lettres!');
  writeln;
  writeln('Les nombres peuvent comporter une partie décimale d''au plus trois');
  writeln('chiffres et ne peuvent comporter plus de six chiffres dans leur partie');
  writeln('entière.');
  writeln('Frappez Entrée pour poursuivre');readln;
  Clrscr;
end;
```

4.7 Analyse de RACCOURCIR

Les spécifications en sont décrites page 172. Il va simplement s'agir de supprimer les 0 terminaux éventuels du nombre entier (compris entre 1 et 999) contenu dans *PartieDecimale* en plaçant ce nombre tronqué dans *PartieRaccourcie*.

4.7.1 RACCOURCIR : Comment faire ?

La stratégie part des constatations suivantes :

- Quand un nombre entre 1 et 999 se termine par 00, c'est qu'il est divisible par 100 et le raccourcir, c'est le diviser par 100
- S'il se termine par un seul 0, c'est qu'il n'est pas divisible par 100, mais seulement par 10. Dans ce cas, le raccourcir, c'est le diviser par 10.

- Dans les autres cas, on laisse le nombre inchangé.

Ce sont bien entendu les fonctions div (qui donne le quotient entier d'une division) et mod (qui donne le reste) qui vont nous permettre de tester la divisibilité et d'effectuer la division entière.

4.7.2 RACCOURCIR : Comment faire faire ?

Simplement :

PartieRaccourcie ← *PartieDecimale* **(1)**
Si *PartieDecimale* divisible par 100 alors
  PartieRaccourcie ← quotient entier de la division entière de *PartieDecimale* par 100
sinon
  si *PartieDecimale* divisible par 10 alors
    PartieRaccourcie ← quotient entier de la division entière de *PartieDecimale* par 10

On notera que le cas où *PartieDecimale* ne se termine pas par 0 aurait pu se traiter en ajoutant un dernier "sinon" avec l'affectation

PartieRaccourcie ← *PartieDecimale*

Pour éviter dans l'écriture Pascal un "else" supplémentaire, j'ai fait figurer cette affectation en **(1)**.

En utilisant les raccourcis possibles grâce à div et mod, on obtient :

PartieRaccourcie ← *PartieDecimale*
Si *PartieDecimale* mod 100 = 0 alors
  PartieRaccourcie ← *PartieDecimale* div 100
sinon
  si *PartieDecimale* mod 10 = 0 alors
    PartieRaccourcie ← *PartieDecimale* div 10

La traduction est immédiate :

4.7.3 RACCOURCIR : Comment dire ?

```
procedure RACCOURCIR;

       (* Elle "raccourcit" éventuellement PartieDecimale en en supprimant les 0 terminaux et place
       le résultat dans PartieRaccourcie *)

begin
PartieRaccourcie := PartieDecimale;
if PartieDecimale mod 100 = 0 then
   PartieRaccourcie := PartieDecimale div 100
else
   if PartieDecimale mod 10 = 0 then
      PartieRaccourcie:=PartieDecimale div 10;
end;
```

4.8 Analyse de FAIRE_SUIVRE_DE_SA_NATURE

Sur base du "Quoi faire ?" rappelé en page 172, on peut débusquer une stratégie.

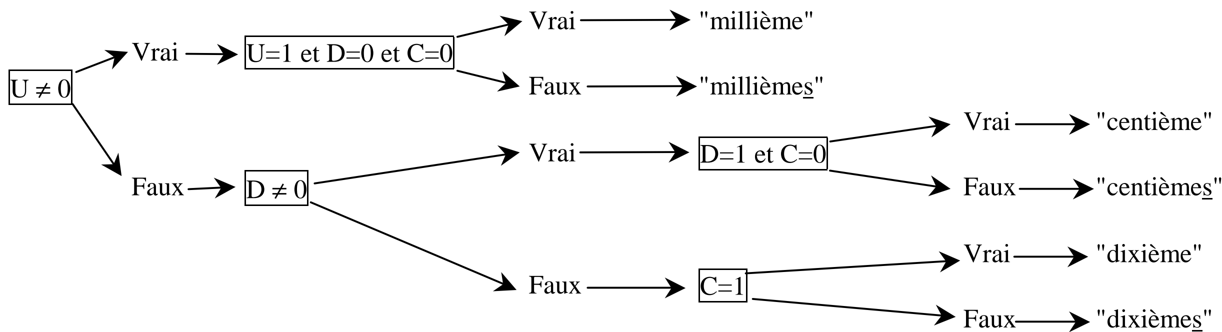
4.8.1 FAIRE_SUIVRE_DE_SA_NATURE : Comment faire ?

Le but est donc de décider quand il faut écrire "dixième(s)", "centième(s)" ou "millième(s)".

C'est à nouveau l'examen du chiffre des unités, de celui des dizaines et de celui des centaines de *PartieDecimale* qui va permettre de résoudre le problème.

- Quand le chiffre des unités n'est pas nul, il faut écrire "millièmes" et "millième" lorsque de plus ce chiffre est 1 et qu'il n'y a ni dizaines, ni centaines.
- Si le chiffre des unités est nul, on a à faire à des "centièmes" ou des "dixièmes" :
  - Quand le chiffre des dizaines n'est pas nul, il faut écrire "centièmes" et même "centième" si ce chiffre est 1 et qu'il n'y a pas de centaine.
  - Quand le chiffre des dizaines est nul, et comme on est dans le cas ou celui des unités est nul aussi, on écrit "dixièmes" et même "dixième" si le chiffre des centaines est 1.

En résumé, disposant de ces trois valeurs, notées U (chiffre des unités), D (chiffre des dizaines) et C (chiffre des centaines), on peut écrire :



Nous voici donc en mesure de fournir la marche à suivre :

4.8.2 FAIRE_SUIVRE_DE_SA_NATURE : Comment faire faire ?

```
ChiffreDesCentaines ← PartieDecimale div 100          ChiffreDesCentaines de type entier
PartieDecimale ← PartieDecimale mod 100    (1)
ChiffreDesDizaines ← PartieDecimale div 10             ChiffreDesDizaines de type entier
ChiffreDesUnités ← PartieDecimale mod 10               ChiffreDesUnités de type entier
Si ChiffreDesUnités ≠ 0 alors
    Si ChiffreDesUnités=1et ChiffreDesDizaines=0 et ChiffreDesCentaines=0 alors
        Affiche 'millième'
    sinon
        Affiche 'millièmes'
sinon
    Si ChiffreDesDizaines ≠ 0 alors
        Si ChiffreDesDizaines=1 et ChiffreDesCentaines=0 alors
            Affiche 'centième'
        sinon
            Affiche 'centièmes'
    sinon
        Si ChiffreDesCentaines=1 alors
            Affiche 'dixième'
        sinon
            Affiche 'dixièmes'
```

Cette marche à suivre semble parfaitement correcte. Et pourtant, si nous relisons les spécifications de FAIRE_SUIVRE_DE_SA_NATURE en page 172, nous constatons que la variable globale *PartieDecimale* peut seulement être consultée par FAIRE_SUIVRE_DE_SA_NATURE et (comme elle ne figure pas à droite du rectangle décrivant le "quoi faire ?") ne peut en aucun cas être modifiée. Et il faut bien convenir que l'affectation notée **(1)** modifie le contenu de cette variable.

Ce n'est évidemment plus le moment de s'interroger sur l'importance réelle pour la marche à suivre de premier niveau et pour les autres modules de la non modification par FAIRE_SUIVRE_DE_SA_NATURE de *PartieDecimale*. Les spécifications de FAIRE_SUIVRE_DE_SA_NATURE interdisaient cette modification : tout écart par rapport aux spécifications est une cause possible d'erreur dans un travail qui se veut modulaire.

Il y a deux manières de circonvenir cette difficulté :

- Nous pouvons définir au sein de FAIRE_SUIVRE_DE_SA_NATURE une variable locale *CopiePartieDecimale* dans laquelle nous commencerons par recopier le contenu de *PartieDecimale*. Nous utiliserons ensuite *CopiePartieDecimale* au lieu de *PartieDecimale*. C'est cette solution que nous allons retenir.
- Nous pourrions également assortir FAIRE_SUIVRE_DE_SA_NATURE d'un paramètre valeur. Au moment de l'appel de FAIRE_SUIVRE_DE_SA_NATURE, nous associerions *PartieDecimale* à ce paramètre. Nous savons que cette solution conduit également à la création d'une variable locale correspondant au paramètre valeur, dans laquelle le contenu de *PartieDecimale* serait copié au moment de l'appel. Mais rien ne justifie par ailleurs la présence de ce paramètre accompagnant FAIRE_SUIVRE_DE_SA_NATURE et comme notre analyse de premier niveau n'avait pas mentionné, dans l'écriture des spécifications de FAIRE_SUIVRE_DE_SA_NATURE ce paramètre, cette solution ne sera pas retenue.

Notons au passage que cette courte discussion illustre bien la différence entre variable globale, variable locale et paramètre valeur.

Avec ces modifications, seul le début de la marche à suivre proposée change pour devenir :

CopiePartieDecimale ← *PartieDecimale*	*CopiePartieDecimale* de type entier
ChiffreDesCentaines ← *CopiePartieDecimale* div 100	*ChiffreDesCentaines* de type entier
CopiePartieDecimale ← *CopiePartieDecimale* mod 100	
ChiffreDesDizaines ← *CopiePartieDecimale* div 10	*ChiffreDesDizaines* de type entier
ChiffreDesUnités ← *CopiePartieDecimale* mod 10	*ChiffreDesUnités* de type entie

On remarquera également que la variable *PartieRaccourcie*, qui était consultable par FAIRE_SUIVRE_DE_SA_NATURE, n'a pas dû être utilisée.

? Pourriez-vous proposer une autre analyse de FAIRE_SUIVRE_DE_SA_NATURE qui se base sur le seul contenu de *PartieRaccourcie* ?

Et nous sommes à présent en mesure d'écrire la procédure correspondante :

4.8.3 FAIRE_SUIVRE_DE_SA_NATURE : Comment dire ?

```
procedure FAIRE_SUIVRE_DE_SA_NATURE;

       (* Elle fait suivre la partie décimale telle qu'elle figure dans PartieDecimale et, après son
       raccourcissement, dans PartieRaccourcie, de sa nature : dixième(s), centième(s) ou
       millième(s) *)

var CopiePartieDecimale,
       (* nécessaire pour ne pas faire les modifications directement dans PartieDecimale   *)
    ChiffreDesCentaines,
    ChiffreDesDizaines,
    ChiffreDesUnites
    (* de la partie décimale *)
    : integer;

begin (* début de FAIRE_SUIVRE_DE_SA_NATURE *)
CopiePartieDecimale :=PartieDecimale;
ChiffreDesCentaines:= CopiePartieDecimale div 100;
CopiePartieDecimale:=CopiePartieDecimale mod 100;
```

```
      (* CopiePartieDecimale contient à présent la partie décimale sans le chiffre des centaines *)
ChiffreDesDizaines:= CopiePartieDecimale div 10;
ChiffreDesUnites:= CopiePartieDecimale mod 10;
if ChiffreDesUnites <> 0 then
   if (ChiffreDesUnites=1) and (ChiffreDesDizaines=0) and (ChiffreDesCentaines=0) then
      write ('millième ')
   else
      write ('millièmes ')
else
   if ChiffreDesDizaines <> 0 then
      if (ChiffreDesDizaines=1) and (ChiffreDesCentaines=0) then
         write ('centième ')
      else
         write ('centièmes ')
   else
      if (ChiffreDesCentaines=1) then
         write ('dixième ')
      else
         write ('dixièmes ');
end;
```

Il nous faut à présent nous attaquer au plat de résistance, la procédure ECRIRE, dont les spécifications sont données page 172.

4.9 Analyse de ECRIRE

Sur base du contenu du paramètre valeur *Nombre*, de type entier long, contenant un entier compris entre 1 et 999999, il faut faire afficher l'écriture en toutes lettres de ce nombre.

4.9.1 ECRIRE : Comment faire ?

L'idée est de scinder le nombre en deux parties : la partie gauche, celle multipliant 1000, et la partie droite, celle inférieure à 1000. Par exemple, dans le cas de 300400, on se retrouverait avec 300 comme partie gauche et 400 comme partie droite.

Il ne resterait plus qu'à écrire en toutes lettres la partie gauche (comprise entre 0 et 999), puis le mot " mille", puis la partie droite (comprise également entre 0 et 999).

Il faut cependant prêter attention aux différences entre l'écriture de la partie gauche (multipliant 1000) et celle de la partie droite (inférieure à 1000). En effet, la règle énoncée page 168 concernant le pluriel de "vingt" et "cent", montre que 200200 s'écrit "deux cent mille deux cents" ou que 80080 s'écrit "quatre vingt mille quatre vingts".

Dans le même ordre d'idées,

- on n'écrit la partie gauche que si elle est strictement supérieure à 1; en effet, pour 1200, on ne dit pas "un mille deux cents", mais "mille deux cents";
- on n'écrit le mot "mille" que si la partie gauche est strictement positive; en effet, pour 800, on dit pas "zéro mille huit cents", mais "huit cents";
- on n'écrit la partie droite que si elle est strictement positive; en effet, pour 3000, on ne dit pas "trois mille zéro", mais seulement "trois mille".

Ces quelques remarques étant faites, on peut passer à l'écriture de la marche à suivre.

4.9.2 ECRIRE : Comment faire faire ?

On peut proposer :

PartieGauche ← *Nombre* div 1000	*PartieGauche* de type entier
PartieDroite ← *Nombre* mod 1000	*PartieDroite* de type entier

Si *PartieGauche* > 1 alors
    ECRIRE_MOINS_DE_MILLE(*PartieGauche*, faux) **(1)**
Si *PartieGauche* > 0 alors
    Affiche 'mille ' **(2)**
Si *PartieDroite* > 0 alors
    ECRIRE_MOINS_DE_MILLE(*PartieDroite*, vrai) **(3)**

avec

ECRIRE_MOINS_DE_MILLE(*NombreInferieurAMille* : entier; *MoinsDeMille* : booléen)

NombreInferieurAMille→ paramètre valeur de type entier
MoinsDeMille→ paramètre valeur de type booléen

Ecrire en toutes lettres le nombre entier *NombreInferieurAMille*. Lorsque *MoinsDeMille* est vrai, "vingt" et "cent" peuvent prendre un s final (en accord avec les règles énoncées). Dans le cas contraire ils ne peuvent en aucun cas s'écrire au pluriel. Enfin tous les mots à écrire seront suivis d'un espace et sans passage à la ligne.

avec

Avant	Après
- *NombreInferieurAMille*, paramètre valeur contenant un nombre entier entre 1 et 999[17] - *MoinsDeMille*, paramètre valeur de type booléen	Comme ces paramètres sont de genre valeur, ils peuvent être modifiés sans problème

J'ai d'emblée utilisé une même procédure pour l'écriture de la partie gauche et de la partie droite du nombre. Le paramètre *NombreInferieurAMille* est indispensable pour que cette unique procédure puisse être appelée d'une part pour l'écriture de *Partiegauche*, d'autre part pour celle de *PartieDroite*. Mais il faut un second paramètre, baptisé *MoinsDeMille*, et qui indiquera à la procédure si la valeur passée est celle de *PartieGauche* ou de *PartieDroite* (pour écrire ou non les éventuelles marques du pluriel à "vingt" et "cent").

On comprend alors que les appels de cette procédure se fassent comme en **(1)** et **(3)** ci-dessus.

- En **(1)** on demande l'écriture de la *PartieGauche* en signalant que *MoinsDeMille* est faux.
- En **(3)** on demande l'écriture de la *PartieDroite* en signalant que *MoinsDeMille* est alors vrai.

On notera que la chaîne 'mille ', dont on demande l'affichage finit par un espace, afin de scinder les constituants du nombre écrit en toutes lettres. On peut à nouveau noter que tous ces affichages se feront évidemment sans passage à la ligne.

4.9.3 ECRIRE : Comment dire ?

```
procedure ECRIRE(Nombre:longint);

       (* Elle fait écrire en toutes lettres Nombre (de type entier long et compris entre 1 et
       999999) Nombre vaudra successivement lors des appels PartieEntiere puis PartieRaccourcie *)

var PartieGauche,
    (* constituant la partie gauche (multipliant 1000) de Nombre *)
    PartieDroite
    (* constituant la partie droite (inférieure à 1000) de Nombre *)
    : integer;

procedure ECRIRE_MOINS_DE_MILLE(NombreInferieurAMille:integer; MoinsDeMille:boolean);

       (* elle fait écrire le contenu de NombreInferieurAMille (qui est forcément un nombre compris
       entre 1 et 999) avec des règles d'écriture un peu différentes pour vingt et cent suivant que
       MoinsDeMille est vrai (ce qui veut dire qu'on écrit la partie inférieure à 1000
       (PartieDroite) de Nombre) ou faux (ce qui signifie qu'on écrit alors la partie gauche
       (PartieGauche), supérieure à 1000 de Nombre). Quand MoinsDeMille est vrai, vingt et cent
```

17 On notera que si *PartieGauche* et *PartieDroite* peuvent être nulles, on n'appelle ECRIRE_MOINS_DE_MILLE que dans le cas où ni *PartieGauche*, ni *PartieDroite*, ne sont nulles.

```
      prennent "s" lorsqu'ils sont multipliés et non suivis; quand MoinsDeMille est faux vingt et
      cent ne prennent pas de "s" *)

begin (* début de ECRIRE *)
PartieGauche:= Nombre div 1000;
PartieDroite:= Nombre mod 1000;
if PartieGauche>1 then
   ECRIRE_MOINS_DE_MILLE(PartieGauche,false);
if PartieGauche>0 then
  write('mille ');
if PartieDroite > 0 then
  ECRIRE_MOINS_DE_MILLE(PartieDroite,true);
end;
```

Il nous reste (à un troisième niveau) à analyser l'action ECRIRE_MOINS_DE_MILLE.

4.10 Analyse de ECRIRE_MOINS_DE_MILLE

Les spécifications sont données page 179. Il faut écrire l'entier (compris entre 1 et 999) contenu dans *NombreInferieurAMille*, avec la terminaison "s" éventuelle à "vingt" et "cent" seulement lorsque *MoinsDeMille* est vrai.

4.10.1 ECRIRE_MOINS_DE_MILLE : Comment faire ?

C'est à nouveau en scindant *NombreInferieurAMille* qu'on peut progresser. Nous allons isoler le nombre de centaines et la partie inférieure à 100. Il suffira alors d'écrire le nombre de centaines, puis "cent " puis la partie inférieure à 100. Comme ci-dessus, il nous faut être attentif à quelques règles :

- On n'écrit le nombre de centaines que s'il est strictement supérieur à 1 : on ne dit pas "un cent...";
- Si le nombre de centaines est 1, on écrit "cent "; s'il est plus grand que 1, on écrit "cents " lorsque MoinsDeMille est vrai et que la partie inférieure à 100 est nulle et "cent " dans le cas contraire.
- On n'écrit la partie inférieure à 100 que si elle est strictement supérieure à 0 : on ne dit pas "trois cents zéro".

4.10.2 ECRIRE_MOINS_DE_MILLE : Comment faire faire ?

On propose donc :

```
NombreDeCentaines ← NombreInferieurAMille div 100          NombreDeCentaines de type entier
NombreInferieurACent ←NombreInferieurAMille mod 100        NombreInferieurACent de type entier
Si NombreDeCentaines > 1 alors
    ECRIRE_LE_NOMBRE_DE_CENTAINES
    Si NombreInferieurACent = 0 et MoinsDeMille alors
        Affiche 'cents '
    sinon
        Affiche 'cent '
Si NombreDeCentaines = 1 alors
    Affiche 'cent '
Si NombreInferieurACent > 0 alors
    ECRIRE_MOINS_DE_CENT
```

avec

	ECRIRE_LE_NOMBRE_DE_CENTAINES
NombreDeCentaines → de type entier	Ecrire en toutes lettres l'entier (compris entre 2 et 9) contenu dans *NombreDeCentaines* (sans passage à la ligne et avec un espace après).

avec

Avant	Après
- *NombreDeCentaines* entier compris entre 2 et 9.	Rien n'a changé

et

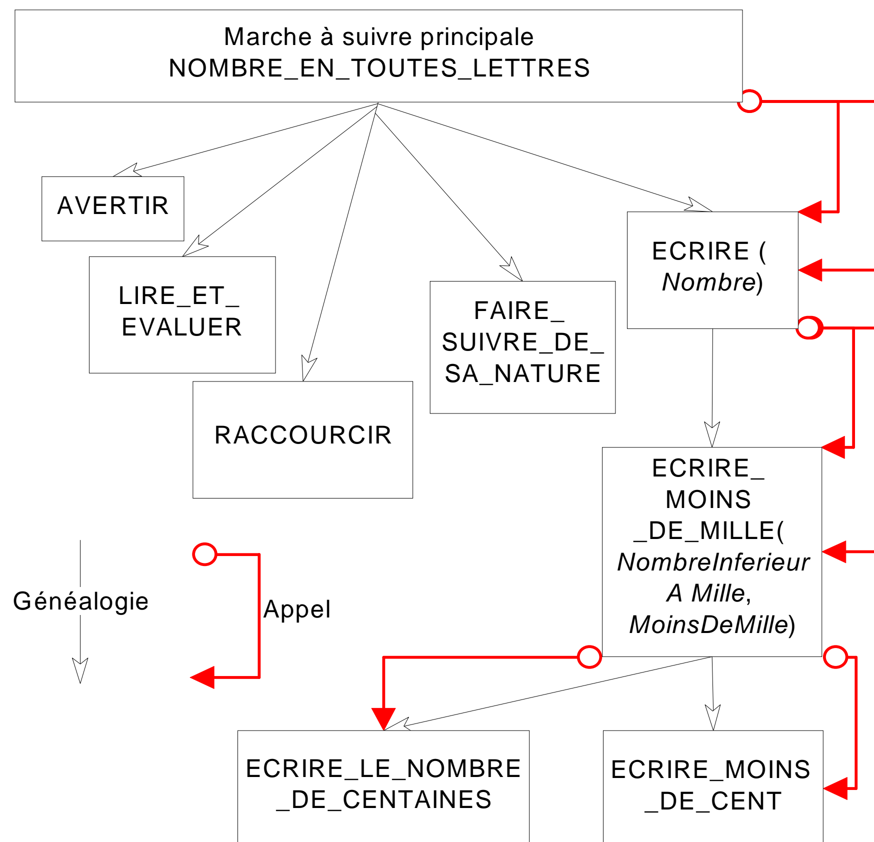
	ECRIRE_MOINS_DE_CENT
NombreInferieurACent → de type entier	Ecrire en toutes lettres l'entier (compris entre 1 et 99) contenu dans *NombreInferieurACent* (sans passage à la ligne et avec un espace après chaque mot).

avec

Avant	Après
- *NombreInferieurACent* entier compris entre 1 et 99.	Rien n'a changé

Les choses sont assez claires. Et pourtant, exceptionnellement, en prévision des remarques qui vont suivre, je ne vais pas immédiatement traduire cette marche à suivre en un programme Pascal.

En faisant le point de la structure actuelle du programme, et en considérant les divers niveaux d'analyse, on a :



On constate à nouveau que seules les procédures *appelées à plusieurs reprises dans des contextes différents* sont obligatoirement assorties des paramètres nécessaires.

Il reste à analyser ECRIRE_MOINS_DE_CENT et ECRIRE_LE_NOMBRE_DE_CENTAINES.

Retenons en tout cas que cette dernière action consiste à écrire un nombre entre 2 et 9.

4.11 Analyse de ECRIRE_MOINS_DE_CENT

Les spécifications en sont données ci-dessus : il faut écrire en toutes lettres la valeur de *NombreInferieurACent*, comprise entre 1 et 99.

4.11.1 ECRIRE_MOINS_DE_CENT : Comment faire ?

Les nombres compris entre 1 et 16 méritent un traitement tout spécial, puisque l'écriture de chacun d'eux ne permet plus aucune décomposition. Notons cependant que parmi ces nombres compris entre 1 et 16, l'écriture de ceux compris entre 2 et 9 réclame exactement le même traitement que celui prévu par l'action ECRIRE_LE_NOMBRE_DE_CENTAINES.

Il semble bien, dès lors, que nous puissions dès à présent définir une action ECRIRE_MOINS_DE_DIX, qui jouera à la fois le rôle assigné à ECRIRE_LE_NOMBRE_DE_CENTAINES mais pourra aussi être appelée par ECRIRE_MOINS_DE_CENT, lors de l'écriture des nombres compris entre 1 et 16.

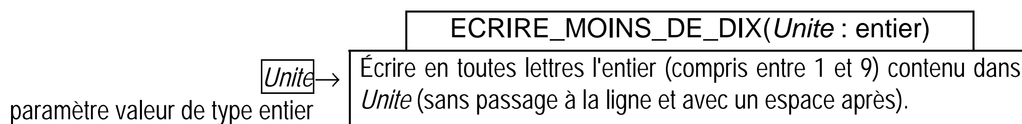
Dans la foulée, on peut prévoir que, même pour les nombres au-delà de 16, ECRIRE_MOINS_DE_CENT devra également faire appel à ECRIRE_MOINS_DE_DIX pour l'écriture du chiffre des unités des nombres entre 17 et 99.

Tout ceci nous conduit à **briser la démarche descendante** pour proposer d'emblée l'écriture d'une procédure ECRIRE_MOINS_DE_DIX, qui, pouvant être appelée par diverses autres procédures, devra être assortie d'un paramètre valeur, de type entier, que nous baptiserons simplement *Unite*.

Quittons donc pour un moment l'analyse de ECRIRE_MOINS_DE_CENT pour définir ECRIRE_MOINS_DE_DIX.

4.12 Analyse de ECRIRE_MOINS_DE_DIX

Les spécifications en sont immédiates :



avec

Avant	Après
- *Unite,* paramètre valeur de type entier contient un nombre entre 1 et 9.	

La structure "Au cas où ..." nous offre évidemment l'outil adapté à l'écriture de cette procédure.

4.12.1 ECRIRE_MOINS_DE_DIX : Comment faire faire ?

```
Au cas où Unite vaut
    1 :     Affiche 'un '
    2 :     Affiche 'deux '
    3 :     Affiche 'trois '
    4 :     Affiche 'quatre '
    5 :     Affiche 'cinq '
```

 © CeFIS - FUNDP 2002

6 : Affiche 'six '
7 : Affiche 'sept '
8 : Affiche 'huit '
9 : Affiche 'neuf '

Ce qui conduit à la procédure suivante :

4.12.2 ECRIRE_MOINS_DE_DIX : Comment dire ?

```
procedure ECRIRE_MOINS_DE_DIX(Unite:integer);

       (* elle fait écrire le nombre Unite compris entre 1 et 9. Elle sera appelée à plusieurs
       reprises : par ECRIRE_ENTRE_17_ET_99, (avec, on le verra, NombreDUnites qui sera alors
       affecté au paramètre Unite) par ECRIRE_MOINS_DE_MILLE (avec NombreDeCentaines affecté à
       Unite),etc. *)

begin
case Unite of
  1 : write('un ');
  2 : write('deux ');
  3 : write('trois ');
  4 : write('quatre ');
  5 : write('cinq ');
  6 : write('six ');
  7 : write('sept ');
  8 : write('huit ');
  9 : write('neuf ');
end;
end;
```

4.13 Retour à l'analyse de ECRIRE_MOINS_DE_MILLE

Comme nous disposons de la procédure ECRIRE_MOINS_DE_DIX, nous pouvons, dans le texte de la procédure ECRIRE_MOINS_DE_MILLE (Cf. page 180), remplacer l'appel de ECRIRE_LE_NOMBRE_DE_CENTAINES par celui de ECRIRE_MOINS_DE_DIX.

4.13.1 ECRIRE_MOINS_DE_MILLE : une nouvelle version du "Comment faire faire ?"

On propose donc :

NombreDeCentaines ← *NombreInferieurAMille* div 100 — *NombreDeCentaines* de type entier
NombreInferieurACent ← *NombreInferieurAMille* mod 100 — *NombreInferieurACent* de type entier
Si *NombreDeCentaines* > 1 alors
  ECRIRE_MOINS_DE_DIX (*NombreDeCentaines*)
  Si *NombreInferieurACent* = 0 et *MoinsDeMille* alors
    Affiche 'cents '
  sinon
    Affiche 'cent '
Si *NombreDeCentaines* = 1 alors
  Affiche 'cent '
Si *NombreInferieurACent* > 0 alors
  ECRIRE_MOINS_DE_CENT

Ceci conduit à la procédure Pascal suivante :

4.13.2 ECRIRE_MOINS_DE_MILLE : Comment dire ?

```
procedure ECRIRE_MOINS_DE_MILLE(NombreInferieurAMille:integer; MoinsDeMille:boolean);

       (* elle fait écrire le contenu de NombreInferieurAMille (qui est forcément un nombre compris
       entre 1 et 999) avec des règles d'écriture un peu différentes pour vingt et cent suivant que
       MoinsDeMille est vrai (ce qui veut dire qu'on écrit la partie inférieure à 1000
```

```
       (PartieDroite) de Nombre) ou faux (ce qui signifie qu'on écrit alors la partie gauche
       (PartieGauche), supérieure à 1000 de Nombre). Quand MoinsDeMille est vrai, vingt et cent
       prennent "s" lorsqu'ils sont multipliés et non suivis; quand MoinsDeMille est faux vingt et
       cent ne prennent pas de "s" *)
var NombreInferieurACent,
    (* partie inférieure à 100 de NombreInferieurAMille *)
    NombreDeCentaines
    (* nombre des centaines de NombreInferieurAMille *)
     : integer;

procedure ECRIRE_MOINS_DE_CENT;
       (* elle fait écrire NombreInferieurACent (compris entre 1 et 99).

begin (* début de ECRIRE_MOINS_DE_MILLE *)
NombreDeCentaines := NombreInferieurAMille div 100;
NombreInferieurACent := NombreInferieurAMille mod 100;
if NombreDeCentaines > 1 then
   begin
   ECRIRE_MOINS_DE_DIX(NombreDeCentaines);
   if (NombreInferieurACent = 0) and MoinsDeMille then
     write('cents ')
    else
     write('cent ');
   end;
 if NombreDeCentaines = 1 then
   write('cent ');
 if NombreInferieurACent > 0 then
   ECRIRE_MOINS_DE_CENT;
 end;
```

4.14 *Retour à l'analyse de ECRIRE_MOINS_DE_CENT*

Disposant, page 181, des spécifications et, connaissant l'existence de la procédure ECRIRE_MOINS_DE_DIX, nous pouvons proposer, suite à l'analyse déjà développée page 182, le texte suivant :

4.14.1 ECRIRE_MOINS_DE_CENT : Comment faire faire ?

Il s'agit donc d'écrire en toutes lettres l'entier *NombreInferieurACent*, compris entre 1 et 99.

Au cas où *NombreInferieurACent* vaut
- 1..9 : [ECRIRE_MOINS_DE_DIX (*NombreInferieurACent*)]
- 10 : Affiche 'dix '
- 11 : Affiche 'onze '
- 12 : Affiche 'douze '
- 13 : Affiche 'treize '
- 14 : Affiche 'quatorze '
- 15 : Affiche 'quinze '
- 16 : Affiche 'seize '
- sinon [ECRIRE_ENTRE_17_ET_99]

avec

	ECRIRE_ENTRE_17_ET_99
NombreInferieurACent → de type entier *MoinsDeMille* → de type booléen	Ecrire en toutes lettres la valeur de *NombreInferieurACent*, sans passage à la ligne et avec un espace après chaque mot. Lorsque *MoinsDeMille* est vrai, 80 s'écrira "quatre-vingts", sinon il s'écrira "quatre-vingt".

avec

Avant	Après
- *NombreInferieurACent* est un entier compris entre 17 et 99 - *MoinsDeMille*, booléen, est vrai si on est en train d'écrire la *PartieDroite* (inférieure à 1000) de Nombre.	- *NombreInferieurACent* n'a pas changé - *MoinsDeMille* n'a pas changé

Nous pouvons immédiatement proposer la procédure Pascal correspondant à ECRIRE_MOINS_DE_CENT.

4.14.2 ECRIRE_MOINS_DE_CENT : Comment dire ?

```
procedure ECRIRE_MOINS_DE_CENT;
       (* elle fait écrire NombreInferieurACent (compris entre 1 et 99 *)

procedure ECRIRE_ENTRE_17_ET_99;
       (* elle fait écrire le nombre NombreInferieurACent compris entre 17 et 99 *)

begin
case NombreInferieurACent of
  1..9 : ECRIRE_MOINS_DE_DIX(NombreInferieurACent);
  10   : write('dix ');
  11   : write('onze ');
  12   : write('douze ');
  13   : write('treize ');
  14   : write('quatorze ');
  15   : write('quinze ');
  16   : write('seize ');
  else
       ECRIRE_ENTRE_17_ET_99;
end;
end;
```

Et pour terminer l'analyse, il ne nous reste plus qu'à nous intéresser à l'action ECRIRE_ENTRE_17_ET_99.

4.15 Analyse de ECRIRE_ENTRE_17_ET_99

Les spécifications en ont été données ci-dessus. On va évidemment continuer à appliquer la stratégie de scission qui a fait ses preuves jusqu'à présent.

4.15.1 ECRIRE_ENTRE_17_ET_99 : Comment faire ?

On considère successivement le nombre D des dizaines et le nombre U des unités de *NombreInferieurACent*.

- L'écriture du mot correspondant aux dizaines est :
  - "dix " quand le chiffre des dizaines vaut 1;
  - "vingt " quand le chiffre des dizaines vaut 2;
  - "trente " quand le chiffre des dizaines vaut 3;
  - "quarante " quand le chiffre des dizaines vaut 4;
  - "cinquante " quand le chiffre des dizaines vaut 5;
  - "soixante " quand le chiffre des dizaines vaut 6;
  - "septante " quand le chiffre des dizaines vaut 7;
  - "nonante " quand le chiffre des dizaines vaut 9;
- Quand le chiffre des dizaines vaut 8, on écrit :
  - "quatre-vingts " quand le chiffre des unités est 0 et que *MoinsDeMille* est vrai

  - "quatre-vingt ", sinon.
- Quand le chiffre des unités est strictement plus grand que 1, on écrit un tiret entre le mot exprimant les dizaines et le mot exprimant les unités. Sinon, quand le chiffre des unités est 1, on écrit "et ", sauf quand le chiffre des dizaines est 8, auquel cas on écrit "quatre-vingt-un".

Avec cette stratégie en tête, on peut passer à l'écriture de la marche à suivre :

4.15.2 ECRIRE_ENTRE_17_ET_99 : Comment faire faire ?

```
NombreDeDizaines ← NombreInferieurACent div 10          NombreDeDizaines de type entier
NombreDUnites ← NombreInferieurACent mod 10             NombreDUnites de type entier
Au cas où NombreDeDizaines vaut
     1 :   Affiche 'dix '
     2 :   Affiche 'vingt '
     3 :   Affiche 'trente '
     4 :   Affiche 'quarante '
     5 :   Affiche 'cinquante '
     6 :   Affiche 'soixante '
     7 :   Affiche 'septante '
     8 :   Si NombreDUnites = 0 et MoinsDeMille alors
                Affiche 'quatre-vingts '
           sinon
                Affiche 'quatre-vingt '
     9 :   Affiche 'nonante '
Si NombreDUnites > 0 alors
     Si NombreDeDizaines = 8 alors
           Ecrire '-'
     sinon
           Si NombreDUnites = 1 alors
               Affiche 'et '
           sinon
               Affiche '-'
     ECRIRE_MOINS_DE_DIX (NombreDUnites)
```

Et nous voici en mesure de terminer avec la procédure correspondante l'analyse de ECRIRE.

4.15.3 ECRIRE_ENTRE_17_ET_99 : Comment dire ?

```
procedure ECRIRE_ENTRE_17_ET_99;
      (* elle fait écrire le nombre NombreInferieurACent compris entre 17 et 99 *)

var NombreDeDizaines,
      (* de NombreInferieurACent *)
    NombreDUnites : integer;
      (* de NombreInferieurACent *)

begin (* Début de ECRIRE_ENTRE_17_ET_99 *)
NombreDeDizaines :=NombreInferieurACent div 10;
NombreDUnites := NombreInferieurACent mod 10;
case NombreDeDizaines of
  1 : write('dix ');
  2 : write('vingt ');
  3 : write('trente ');
  4 : write('quarante ');
  5 : write('cinquante ');
  6 : write('soixante ');
  7 : write('septante ');
  8 : if (NombreDUnites = 0) and MoinsDeMille then
```

```
        write('quatre-vingts ')
       else
        write('quatre-vingt ');
  9 : write('nonante ');
end;
if NombreDUnites > 0 then
  begin
  if NombreDeDizaines = 8 then
    write('-')
   else
    if NombreDUnites=1 then
      write('et ')
     else
      write('-');
  ECRIRE_MOINS_DE_DIX(NombreDUnites);
  end;
end;
```

L'essentiel est terminé : il reste encore à analyser LIRE_ET_EVALUER dont les spécifications sont données page 171. Comme cette analyse n'apportera aucun élément nouveau et comme toute la discussion qui reste à mener porte sur la position qu'occupera ECRIRE_MOINS_DE_DIX dans la généalogie des procédures, le lecteur peut passer directement à la page 192, en revenant dans la suite à l'analyse de LIRE_ET_EVALUER.

4.16 Analyse de LIRE_ET_EVALUER

4.16.1 LIRE_ET_EVALUER : Comment faire ?

Le point le plus important est le choix de la forme sous laquelle le nombre fourni par l'utilisateur sera codé; on sait que si l'on choisit de considérer la donnée comme un réel, aucune vérification du fait que les caractères sont bien des chiffres n'est possible : ou ce que l'utilisateur fournit est bien l'écriture d'un réel et la donnée est acceptée, ou ce n'est pas le cas et le programme "se plante". L'unique solution consistera forcément à lire le nombre fourni par l'utilisateur comme une chaîne de caractères et à utiliser à bon escient la procédure de transformation d'une chaîne en un nombre. L'essentiel de la démarche va donc consister à lire la chaîne de caractères fournie par l'utilisateur et à explorer cette dernière pour en extraire la sous-chaîne (éventuellement vide) reprenant la partie entière du nombre et la sous-chaîne (éventuellement vide) représentant sa partie décimale. En cas d'erreur, la lecture sera reprise.

Les étapes seront donc les suivantes :

- Lire comme une chaîne de caractères le nombre fourni par l'utilisateur.
- Si la longueur de cette chaîne est nulle ou dépasse 10, il y a une erreur.
- En l'absence d'erreur, on va explorer la chaîne, caractère après caractère, en bâtissant chiffre après chiffre, la chaîne constituant la partie entière (qui peut être vide) puis la chaîne constituant la partie décimale (qui peut être vide également); tout écart par rapport à l'écriture correcte d'un réel et par rapport aux spécifications (partie entière comportant au plus 6 caractères-chiffres, partie décimale en comportant au plus 3) sera signalé comme une erreur.
- Tout ce processus (lecture, validation, décomposition) sera repris jusqu'à ce que plus aucune erreur n'apparaisse.
- La chaîne constituant la partie entière, puis celle constituant la partie décimale (complétée par des 0 terminaux pour avoir 3 caractères, puisqu'on veut la partie décimale exprimée en millièmes) seront alors transformées grâce à la procédure <u>val</u> (Cf. p. 211) en des nombres

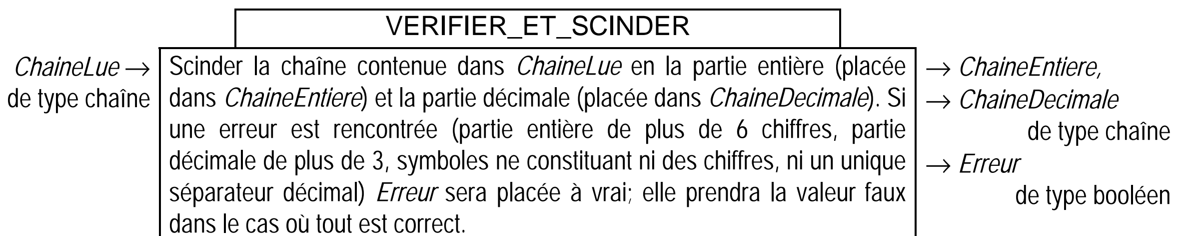
entiers.

4.16.2 LIRE_ET_EVALUER : Comment faire faire ?

La connaissance des outils de traitement des chaînes de caractères (string) est ici essentielle. On consultera pa exemple l'annexe page 207 à ce propos.

Répéter
    Erreur ← faux — *Erreur* de type booléen
    Lis et place dans *ChaineLue* — *ChaineLue* de type string[20]
    VERIFIER_ET_SCINDER — *ChaineEntiere* de type string[6], *ChaineDecimale* de type string[3]
jusqu'à ce que pas *Erreur*
Si *ChaineEntiere* est vide alors **(1)**
    ChaineEntiere ← '0'
Tant que longueur de *ChaineDecimale* < 3 **(2)**
    Accoller '0' au bout de *ChaineDecimale*
Transformer *ChaineEntiere* en l'entier *PartieEntiere* **(3)**
Transformer *ChaineDecimale* en l'entier *PartieDecimale*

avec



avec

Avant	Après
- *ChaineLue,* de type chaîne (string[20]), au contenu quelconque, fourni par l'utilisateur. - *ChaineEntiere,* de type chaîne (string[6]), au contenu quelconque. - *ChaineDecimale,* de type chaîne (string[3]), au contenu quelconque. - *Erreur,* de type booléen contient faux.	- *ChaineLue,* non modifiée - *ChaineEntiere,* de type chaîne, contient la succession des caractères-chiffres constituant la partie entière du nombre (au plus 6 caractères et éventuellement vide) pour autant que *Erreur* soit faux; sinon (si *Erreur* est vrai) le contenu de *ChaineEntiere* est quelconque. - *ChaineDecimale,* de type chaîne, contient la succession des cracatères-chiffres constituant la partie décimale du nombre (au plus 3 caractères et éventuellement vide) pour autant que *Erreur* soit faux; sinon (si *Erreur* est vrai) le contenu de *ChaineDecimale* est quelconque. - *Erreur,* de type booléen, vrai si une erreur ou une non conformité aux spécifications a été détectée dans l'écriture du nombre contenu dans *ChaineLue.*

Quelques remarques sont indispensables :

(1) En l'absence de précisions sur le comportement de la procédure Pascal val chargée de transformer une chaîne en un nombre, j'ai préféré, lorsque la chaîne *PartieEntiere* est vide, ne pas risquer d'erreur en plaçant alors explicitement dans *PartieEntiere* le caractère "0".

Il semble bien (en tout cas dans certaines implémentations de Pascal) que dans le cas d'une chaîne vide, val fournisse comme nombre correspondant 0, ce qui rend inutile le test et l'éventuelle affectation du caractère "0" à *ChaineEntiere*.

(2) Le nombre constituant la partie décimale devra exprimer des millièmes. Il faut donc que la chaîne *ChaineDecimale* soit constituée de chiffres tels que l'application de val à cette chaîne fournisse bien le nombre de millièmes. Ainsi, si *ChaineLue* contenait "0.9", la chaîne *ChaineDecimale* contiendrait le caractère "9". L'application de val fournirait alors dans *PartieDecimale* l'entier 9, alors que c'est 900 (millièmes) qui devrait y figurer. Le fait de compléter *ChaineDecimale* par des 0 terminaux (jusqu'à avoir une chaîne de 3 caractères), transformera, dans l'exemple évoqué, cette chaîne en "900"; dans ce cas val fournira bien la valeur attendue.

Notons au passage que nous postulons de toute manière un comportement adéquat de la procédure val lorsque *ChaineDecimale* contient des 0 initiaux. On pourrait en effet, si *ChaineLue* contient "0.09", retrouver à l'issue de VERIFIER_ ET_SCINDER une *ChaineDecimale* contenant "09", puis "090" après complétion par un "0" terminal. La procédure val néglige heureusement le "0" initial et place bien dans *PartieDecimale* le nombre 90.

(3) C'est, je l'ai déjà signalé, la procédure val qui se charge de la transformation d'une chaîne en nombre (entier ou réel). Cette procédure comporte en réalité 3 paramètres (Cf. page 211) :

- le premier, de type chaîne et de genre valeur, accueille en entrée la chaîne à transformer en nombre;
- le second, de type entier ou réel et de genre variable, fournit dans la variable qui y sera associée le nombre correspondant à la chaîne fournie dans le premier paramètre;
- le troisième, de type entier et de genre variable, fournit dans la variable entière associée une valeur nulle lorsque la transformation a pu être menée à bien (parce que la chaîne fournie dans le premier paramètre représente bien un nombre, entier ou réel, correctement écrit). Lorsque la valeur de ce troisième paramètre (variable) est non nulle, elle désigne le numéro du premier caractère à partir duquel la chaîne passée comme premier paramètre n'est plus l'écriture correcte d'un entier ou d'un réel.

En ce qui nous concerne, l'exécution préalable de VERIFIER_ET_SCINDER assure que *ChaineEntiere* et *ChaineDecimale* représentent bien des nombres. Dès lors la vérification effectuée par val est inutile. Il faut pourtant bien disposer d'une variable qui sera associée au troisième paramètre de val, même si cette variable est inutilisée.

Dès lors, les deux dernières instructions de la marche à suivre proposée vont devenir :

val(*ChaineEntiere, PartieEntiere, NeSertArien*) *NeSertARien* de type entier
val(*ChaineDecimale, PartieDecimale, NeSertARien*)

Et nous voilà, une fois de plus en mesure de fournir la procédure correspondante.

4.16.3 LIRE_ET_EVALUER : Comment dire ?

```
procedure LIRE_ET_EVALUER;

        (* Elle fait lire un nombre (sous la forme d'une chaîne de caractères) et vérifie qu'il a
        bien les caractéristiques voulues ( au plus 6 chiffres pour la partie entière et 3 chiffres
        pour la partie décimale. Elle fait placer la succession des chiffres de la partie entière
        dans PartieEntiere et la succession des chiffres de la partie décimale dans PartieDecimale
        par appel de la procédure VERIFIER_ET_SCINDER. Les erreurs éventuelles sont détectées et la
        lecture est alors recommencée *)

var ChaineLue : string[25];
        (* succession des caractères du nombre à écrire *)
    ChaineEntiere : string[6];
        (* succession des chiffres de la chaîne constituant la partie entière *)
    ChaineDecimale : string[3];
        (* succession des chiffres de la chaîne constituant la partie décimale *)
    NeSertARien
    (* variable entière nécessaire pour l'appel de la procédure val *)
```

```
    : integer;
    Erreur : boolean;

procedure VERIFIER_ET_SCINDER;

       (* Elle explore ChaineLue et ajoute les caractères successivement extraits au bout de
       ChaineEntiere ou de ChaineDecimale selon qu'on est en train d'explorer la partie entière du
       nombre ou sa partie décimale. En cas d'erreur, le booléen Erreur est placé à vrai. *)

begin
repeat
  Erreur:=false;
  Clrscr;
  write('Donnez le nombre à écrire : ');
  readln(ChaineLue);
  VERIFIER_ET_SCINDER;
until not Erreur;
if ChaineEntiere = '' then
  ChaineEntiere:='0';
       (* on complète éventuellement la partie décimale par des 0 pour obtenir un "nombre" (une
       chaîne) à 3 chiffres *)
while length(ChaineDecimale) < 3 do
  ChaineDecimale:= ChaineDecimale + '0'; (* + sert à concaténer les chaînes *)
val(ChaineEntiere,PartieEntiere,NeSertArien);
val(ChaineDecimale,PartieDecimale,NeSertARien);
end;
```

4.17 Analyse de VERIFIER_ET_SCINDER

Les spécifications sont fournies page 188.

4.17.1 VERIFIER_ET_SCINDER : Comment faire ?

Voilà un beau petit problème de programmation (comme en fournissent très souvent les manipulations à propos des chaînes de caractères).

D'abord, lorsque la longueur du nombre à explorer est nulle ou supérieure à 10, il y a certainement une erreur dans l'écriture du nombre et on peut donc en rester là.

Sinon, dans le parcours de la chaîne à explorer, deux signaux vont être indispensables : l'un, déjà connu et disponible est *Erreur* qui permettra de signaler qu'un problème a été détecté; il en faudra un second qui signale qu'on est toujours en train d'explorer la partie entière ou bien que, ayant déjà rencontré le séparateur décimal (point ou virgule), on est en train d'explorer la partie décimale.

L'exploration s'arrêtera donc lorsqu'on arrive au dernier caractère du nombre ou dès qu'une erreur est signalée. Un caractère étant examiné, plusieurs cas peuvent se présenter :

- Le caractère est bien un chiffre et on est toujours dans la partie entière du nombre; dans ce cas, de deux choses l'une :
  - ou bien la chaîne représentant la partie entière n'a pas encore atteint 6 chiffres, et on y adjoint donc le caractère examiné,
  - ou bien cette taille est atteinte, et dans ce cas, la partie entière est trop longue et une erreur doit être signalée.
- Le caractère est bien un chiffre mais on n'est plus dans la partie entière; dans ce cas, de deux choses l'une :
  - ou bien la chaîne représentant la partie décimale n'a pas encore atteint 3 chiffres, et on y adjoint donc le caractère examiné,

  - ou bien cette taille est atteinte, et dans ce cas, la partie décimale est trop longue et une erreur doit être signalée.
- Le caractère rencontré est le séparateur décimal. Dans ce cas,
  - ou bien le signal atteste qu'on est toujours dans la partie entière, et il faut simplement le faire basculer pour tenir compte de la rencontre du séparateur décimal;
  - ou bien on n'était déjà plus dans la partie entière et une erreur doit être signalée (puisque dans ce cas le séparateur décimal est plusieurs fois présent dans le "nombre" examiné).

Il va évidemment falloir organiser la boucle d'exploration de la chaîne à examiner et structurer correctement ces trois alternatives, mais nous tenons là une stratégie raisonnable.

4.17.2 VERIFIER_ET_SCINDER : Comment faire faire ?

```
Initialiser ChaineEntiere à la chaîne vide
Initialiser ChaineDecimale à la chaîne vide
LongueurDuNombre ← longueur de ChaineLue                    LongueurDuNombre de type entier
Erreur←(LongueurDuNombre =0 ou LongueurDuNombre > 10)
OnEstDansLaPartieEntiere ← vrai                        OnEstDansLaPartieEntiere de type booléen
I ← 0                                                                    I de type entier
Tant que pas Erreur et I < LongueurDuNombre
    I ← I+1
    C ← caractère n° I de ChaineLue                                      C de type caractère
    Si C ≥ '0' et C ≤ '9' et OnEstDansLaPartieEntiere alors
        Si longueur de ChaineEntiere<6 alors
            ChaineEntiere ← ChaineEntiere + C
        sinon
            Erreur ← vrai
    sinon
        Si C ≥ '0' et C ≤ '9' et  pas OnEstDansLaPartieEntiere alors
            Si longueur de ChaineDecimale<6 alors
                ChaineDecimale ← ChaineDecimale + C
            sinon
                Erreur ← vrai
        sinon
            Si (C='.' ou C=',') et OnEstDansLaPartieEntiere  alors
                OnEstDansLa PartieEntiere ← faux
            sinon
                Erreur ← vrai
```

La marche à suivre proposée se comprend aisément. Il est important de se convaincre que cette gestion des alternatives multiples rend bien compte de la stratégie mise au jour.

? Pouvez-vous proposer d'autres formes possibles pour les alternatives nécessaires ?

Nous voilà en mesure de terminer avec le texte de la procédure correspondante.

4.17.3 VERIFIER_ET_SCINDER : Comment dire ?

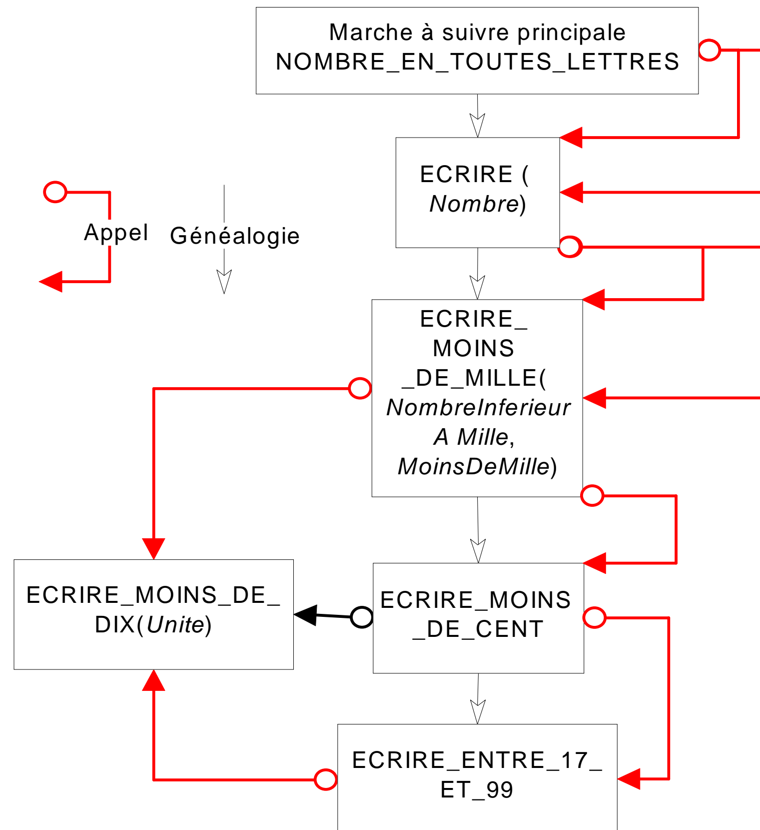
```
procedure VERIFIER_ET_SCINDER;
```

```
       (* Elle explore ChaineLue et ajoute les caractères C successivement extraits au bout de
       ChaineEntiere ou de ChaineDecimale selon qu'on est en train d'explorer la partie entière du
       nombre ou sa partie décimale. En cas d'erreur, le booléen Erreur est placé à vrai. *)

var C:char;
       (* qui contiendra chaque caractère extrait de ChaineLue *)
    I,
       (* compteur *)
    LongueurDuNombre
       (* longueur de la chaîne lue *)
    : integer;
    OnEstDansLaPartieEntiere: boolean;
       (* vrai lorsque l'on est en train d'explorer la partie entière du nombre donc on n'a pas
       encore rencontré la virgule ou le point; faux après cette rencontre *)

begin
LongueurDuNombre := length(ChaineLue);
Erreur:= (LongueurDuNombre=0)or(LongueurDuNombre>10);
(* initialisations *)
OnEstDansLaPartieEntiere:=true;
I:=0;
ChaineEntiere:='';
ChaineDecimale:='';
while not(Erreur) and (I<LongueurDuNombre) do
  begin
  I:=I+1;
  C:= ChaineLue[I];
  if (C>='0')and(C<='9') and OnEstDansLaPartieEntiere then
     if length(ChaineEntiere)<6 then
      ChaineEntiere:=ChaineEntiere+C
     else
      Erreur:=true
  else
     if (C>='0')and(C<='9') and not OnEstDansLaPartieEntiere then
        if length(ChaineDecimale)<3 then
         ChaineDecimale:=ChaineDecimale+C
        else
         Erreur:=true
     else
        if ((C='.')or(C=',')) and OnEstDansLaPartieEntiere then
          OnEstDansLaPartieEntiere:=false
        else
          Erreur:=true;
   end;
 end;
```

Nous voici au terme de l'analyse de ce problème de chiffres et de lettres. Il nous faut cependant revenir à présent sur la question de l'organisation de la généalogie des procédures.

4.18 L'organisation générale du programme global

Dans le schéma suivant, j'ai ignoré les diverses procédures de premier niveau pour lesquelles aucun problème ne se pose : LIRE_ET_EVALUER, AVERTIR, FAIRE_SUIVRE_DE_SA_NATURE et RACCOURCIR pour focaliser l'attention sur ECRIRE et les procédures appelées dans le cadre de l'exécution de cette dernière.



- Pas de problème nouveau à signaler en ce qui concerne les deux appels de ECRIRE par le programme principal : ECRIRE est fille du programme principal et le paramètre dont elle est assortie permet les deux appels successifs.
- Pas de problème non plus en ce qui concerne les deux appels de ECRIRE_MOINS_DE_MILLE par ECRIRE : ECRIRE_MOINS_DE_MILLE est fille de ECRIRE et les paramètres dont elle est munie permettent les deux appels.
- Aucun problème pour ECRIRE_MOINS_DE_CENT : fille de ECRIRE_MOINS_DE_MILLE, elle est appelée une seule fois par cette dernière (et ne nécessite donc pas de paramètre).
- Même chose pour ECRIRE_ENTRE_17_ET_99 : fille de ECRIRE_MOINS_DE_CENT, elle est appelée une seule fois par cette dernière.
- Le problème se pose évidemment pour ECRIRE_MOINS_DE_DIX : elle est appelée à la fois par ECRIRE_MOINS_DE_MILLE, par ECRIRE_MOINS_DE_CENT et par ECRIRE_ENTRE_17_ET_99. Il est donc normal qu'elle soit assortie d'un paramètre auquel seront associées des valeurs différentes lors des trois appels.

  Mais de qui ECRIRE_MOINS_DE_DIX doit-elle être la fille si nous voulons qu'elle puisse être appelée par les trois procédures mentionnées ?

  Et pour formuler de manière plus générale cette question : une procédure peut-elle appeler d'autres procédures que ses filles ? Ou encore : "qui peut appeler qui ?" dans la généalogie que dessinent les diverses procédures. C'est à cette question que je vais à présent répondre avant de pouvoir décider de qui ECRIRE_MOINS_DE_DIX sera la fille.

5. Les procédures : qui peut appeler qui ?

5.1 Les appels possibles

Il va de soi que la réponse apportée à cette question est celle propre au langage Pascal.

L'essentiel, comme souvent, n'est pas la réponse mais la question elle-même : elle fournit en quelque sorte un élément de la grille d'analyse qu'on peut appliquer à tout langage. Bien d'autres questions sont d'ailleurs pertinentes : quelle est la portée des variables ? de quel genre de paramètre dispose-t-on ? etc. dans tel langage.

Avec le vocabulaire aux connotations généalogiques employé ici, la réponse est rapide :

1. Comme nous le savons déjà, une procédure (et le programme principal) peut appeler n'importe laquelle de ses filles.
2. Une procédure peut également s'appeler elle-même : c'est toute la pensée récursive qui est au coeur de cette possibilité. Nous n'utiliserons pas ici cette virtualité qui débouche sur une tout autre méthodologie d'analyse et de programmation.

   Notons que cette possibilité n'est cependant pas valable pour le programme principal qui ne peut être appelé par personne (y compris par lui-même).
3. Une procédure peut appeler sa mère et n'importe laquelle de ses ascendantes (sauf le programme principal).
4. Une procédure peut appeler l'une quelconque de ses soeurs aînées. Par soeur aînée, j'entends ici une procédure ayant la même procédure-mère mais dont le texte **précède** celui de la procédure appelante.
5. Une procédure peut appeler n'importe quelle soeur aînée de n'importe laquelle de ses ascendantes.

On notera par contre que :

a) Une procédure ne peut appeler sa petite-fille, ni aucune autre descendante au delà du premier degré.

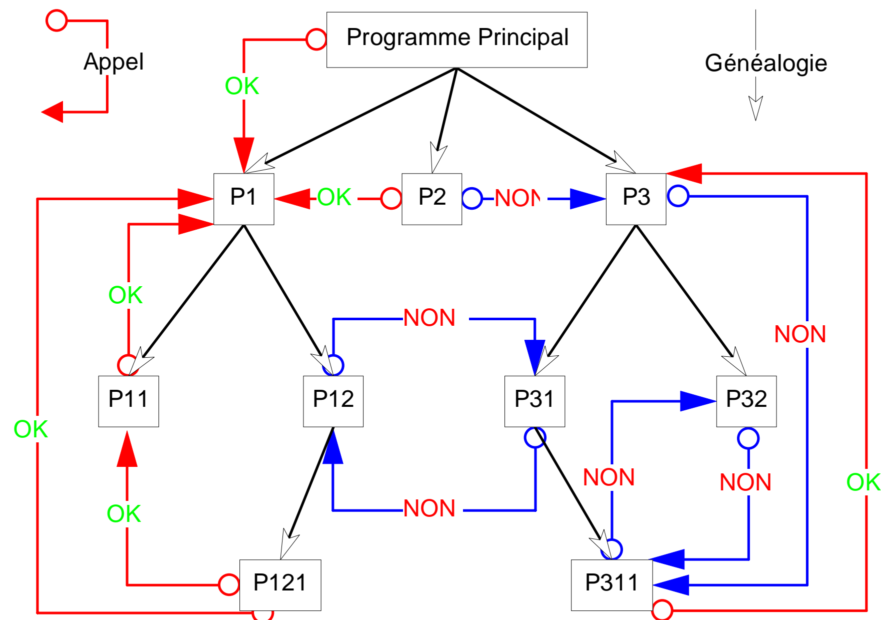
b) Une procédure ne peut appeler une cousine.

c) Une procédure ne peut appeler une soeur cadette (ceci sera corrigé plus bas, avec la possibilité d'une déclaration "forward").

d) Une procédure ne peut appeler les soeurs cadettes de ses ascendantes.

e) Une procédure ne peut appeler aucune descendante de ses soeurs; ainsi une procédure ne peut appeler une nièce

Ainsi, dans le tableau suivant, où à un même niveau de procédures soeurs, on écrit les procédures aînées les plus à gauche :



On constate par exemple que :

- Le programme principal peut appeler sa fille P1 (règle 1);
- P2 peut appeler sa soeur aînée P1 (règle 4);
- P2 ne peut appeler sa soeur cadette P3 (règle c);
- P3 ne peut appeler sa petite fille P311 (règle a);
- P11 peut appeler sa mère P1 (règle 3);
- P12 ne peut appeler sa cousine P31 (règle b);
- P31 ne peut appeler sa cousine P12 (règle b);
- P32 ne peut appeler sa nièce P311 (règle e);
- P121 peut appeler sa tante (soeur aînée de sa mère) P11 (règle 5);
- P121 peut appeler sa grand-mère P1 (règle 3);
- P311 ne peut appeler sa tante (soeur cadette de sa mère) P32 (règle d);
- P311 peut appeler sa grand-mère P3 (règle 3).

5.2 Appels de procédures et portée des variables

Comme on le voit, les possibilités d'appels entre procédures sont fort riches et fort complexes.

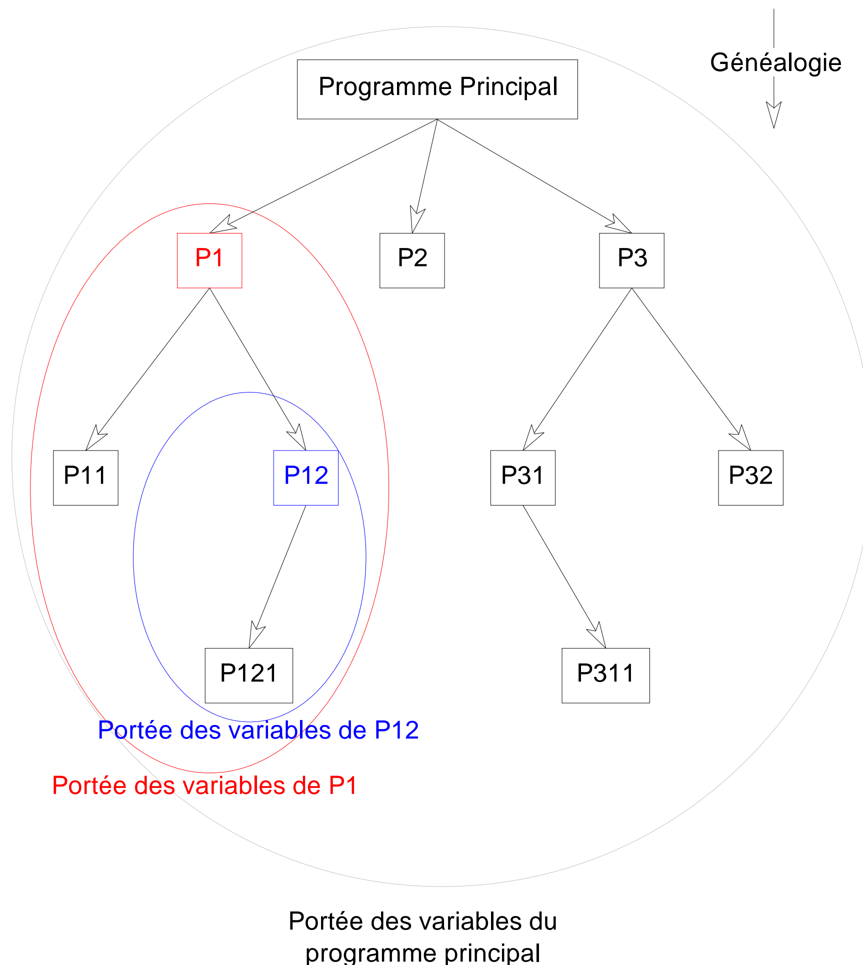
D'autre part, j'ai déjà eu l'occasion d'aborder (au chapitre 2) le concept de portée d'une variable ou d'une déclaration (de constante, de type,...).

Dans le cas d'une approche descendante strictement comprise, les 2 règles qui étaient de mise étaient parfaitement cohérentes :

- Une procédure (ou le programme principal) n'appelle que ses procédures filles; chacune de ces filles n'est d'ailleurs appelée qu'une seule fois (et ne nécessite donc aucun paramètre).

- Les variables définies au sein d'une procédure (et plus généralement les déclarations de constante, de type,...) sont accessibles à toutes les descendantes.

Ainsi, n'importe quelle procédure dispose de toutes les variables installées par la lignée de ses ascendantes, chacune des ces ascendantes ayant appelé une de ses filles et ayant peu à peu conduit à l'appel de la procédure concernée.



Ainsi, dans le schéma ci-dessus, la procédure P121 dispose des variables de toute la lignée qui a abouti à son appel : celles du programme principal, celles de la procédure P1 (sa grand-mère), celles de la procédure P12 (sa mère) et les siennes (variables locales de P121).

En quelque sorte, les variables disponibles suivent les appels effectués, ces appels collant parfaitement à la généalogie existante.

Cette règle de portée des variables est évidemment battue en brèche lors des **appels qui sortent de la stricte généalogie**. On a vu qu'une procédure peut appeler une procédure soeur aînée. Mais bien évidemment la soeur aînée en question, si elle peut accéder aux variables de toutes ses ascendantes ne peut accéder aux variables de la soeur qui l'appelle.

Ainsi, dans le schéma ci-dessus, P12 peut appeler sa soeur P11; mais la portée des variables de P12 est limitée à cette procédure P12 et à sa fille P121. Dès lors, P11 (appelée) ne peut accéder aux variables installées par P12 (appelante).

Dès lors les explications données au chapitre 2, assurant qu'une procédure accédait à toutes les variables installées jusque là, n'est valable que dans une démarche descendante où tous les

 © CeFIS - FUNDP 2002

appels ne se font qu'entre mère et fille. Dans tous les autres cas, même si des variables ont été installées, une procédure ne peut accéder qu'aux variables installées par ses ascendantes et non aux variables installées par d'autres.

La solution dans ce cas est évidemment immédiate : la procédure appelée par une autre procédure que sa mère doit être assortie des paramètres nécessaires : toutes les entrées nécessaires à cette procédure et toutes les sorties vont transiter par ces paramètres.

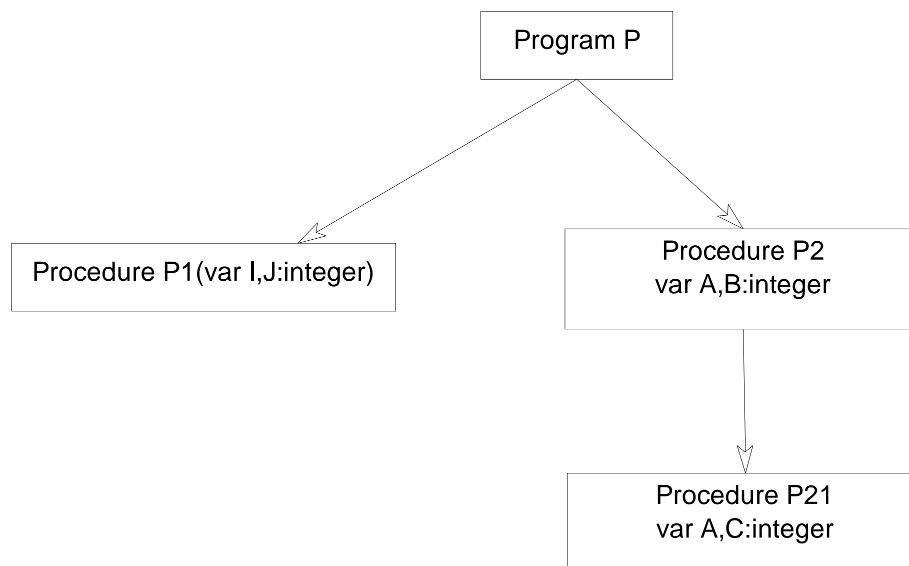
Les paramètres seront donc utilisés dans deux cas :

- Dans le cas où un appel traditionnel mère → fille est effectué plusieurs fois, le travail de la procédure fille se faisant à chaque appel à partir de valeurs et sur des variables différentes. C'est le cas ci-dessus pour la procédure ECRIRE qui appelle à deux reprises sa fille ECRIRE_MOINS_DE_MILLE.
- Dans le cas où l'appel n'est plus entre mère et fille; dans ce cas la procédure appelée peut n'avoir aucun accès direct aux variables de l'appelante : seuls les paramètres permettent alors de mettre à disposition de l'appelée les valeurs, à travers les paramètres valeurs, et les variables nécessaires, à travers des paramètres variables.

  Notons d'ailleurs que dans ce cas, la procédure appelée est de toute manière généralement appelée au moins à deux reprises : par sa mère et par une autre procédure.

  Ce sera le cas de la procédure ECRIRE_MOINS_DE_DIX appelée à trois reprises par 3 procédures différentes (une seule des trois pouvant être sa mère).

Lorsqu'une procédure en appelle une autre, cette dernière étant assortie des paramètres nécessaires, elle peut associer à ces divers paramètres n'importe laquelle des variables auxquelles elle a accès. Ainsi dans l'exemple suivant :



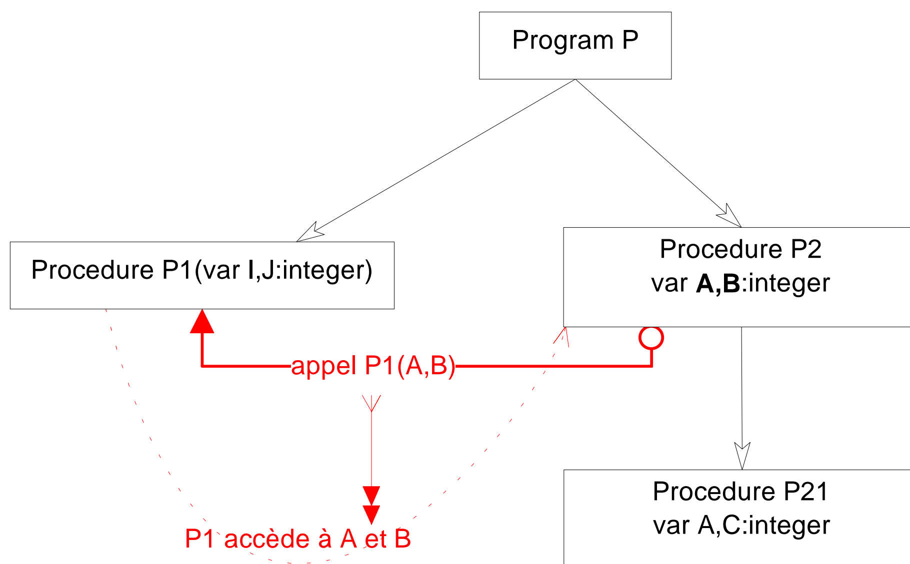
- la procédure P1 est assortie de deux paramètres I et J de type entier qui sont des paramètres variables, mais le raisonnement serait pratiquement identique pour des paramètres valeurs;
- la procédure P2 , soeur cadette de P1, possède deux variables entières locales, A et B;
- la procédure P21, fille de P2, possède deux variables entières locales, A et C.

  En vertu des règles d'appel énoncées,

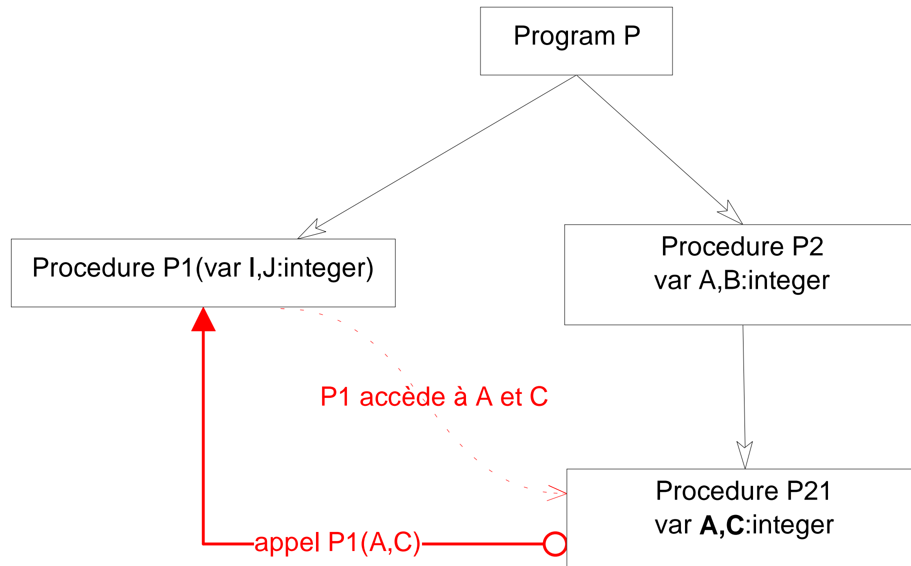
- P2 peut appeler sa soeur aînée P1, mais P1 ainsi appelée n'a pas accès directement aux variables A et B définies au sein de P2
- P21 peut appeler sa tante P1, mais P1 ainsi appelée n'a pas accès directement aux variables A et C définies au sein de P21

Par contre, comme P1 possède deux paramètres I et J, ces deux paramètres formels doivent se voir associer des paramètres effectifs lors des appels de P1. Ainsi

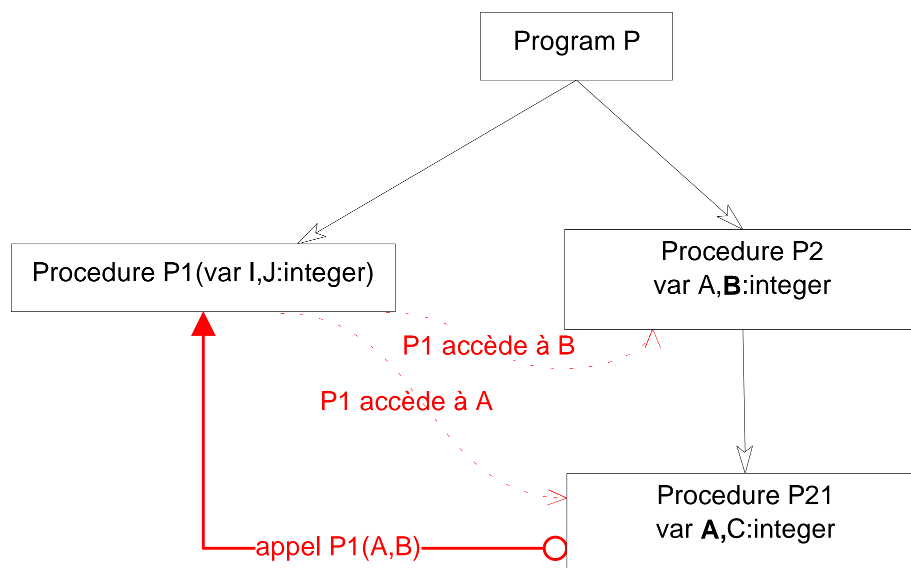
- Au sein de P2, on peut trouver un appel à P1 du style P1(A,B); ceci signifie que le paramètre I donne accès à la variable A et J à la variable B. Dans ce cas, par l'intermédiaire des paramètres, P1 a bien accès, indirectement en quelque sorte, aux variables A et B de P2.



- Au sein de P21, on peut trouver un appel à P1 du style P1(A,C); ceci signifie que le paramètre I donne accès à la variable A de P21 et J à la variable C de P21. Dans ce cas, par l'intermédiaire des paramètres, P1 a bien accès, indirectement en quelque sorte, aux variables A et C de P21. Il faut remarquer que, comme c'est P21 qui appelle P1, le sens à accorder à la mention A est celui de la variable A "la plus proche" de P21; dans ce cas c'est la variable A locale à P21; même si une variable A existe aussi chez P2, lorsque P21 parle de A, c'est de "sa" variable A qu'il s'agit.



- Au sein de P21, on peut trouver un appel à P1 du style P1(A,B); ceci signifie que le paramètre I donne accès à la variable A de P21 et J à la variable B de P2, mère de P21. Dans ce cas, par l'intermédiaire des paramètres, P1 a bien accès, indirectement en quelque sorte, à la variable A de P21 et à la variable B de P2, mère de P21. Il faut remarquer que, comme c'est P21 qui appelle P1, le sens à accorder à la mention A est celui de la variable A "la plus proche" de P21; dans ce cas c'est la variable A locale à P21; même si une variable A existe aussi chez P2, lorsque P21 parle de A, c'est de "sa" variable A qu'il s'agit. Par contre la variable B associée à J est celle que P21 connaît parce que c'est une variable de sa mère P2.

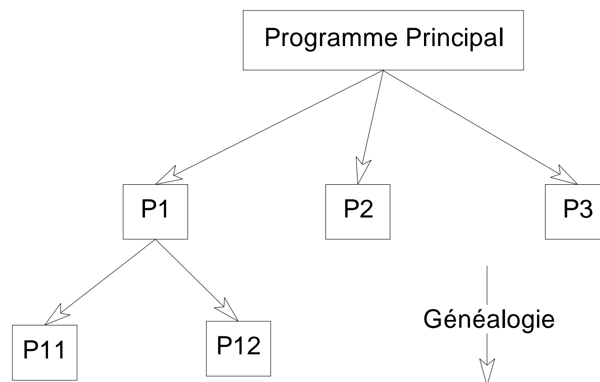


Lors de l'appel d'une procédure possédant des paramètres, n'importe quelle variable *accessible à la procédure appelante*, peut être associée à ces paramètres. Le fait que ces variables ne soient pas accessible directement par la procédure appelée n'a pas d'importance : ce qui compte lors de l'appel faisant usage des paramètres, ce sont les variables auxquelles la procédure appelante a accès.

5.3 *La possibilité du FORWARD*

On sait qu'une procédure peut appeler n'importe quelle soeur aînée, c'est à dire n'importe quelle procédure logée dans la même procédure mère, mais dont le texte précède.

Ainsi, si le schéma de la généalogie des procédures est



correspondant à une structure de programme :

```
Program Principal;
Déclarations du programme principal

        procedure P1
        Déclarations de P1


                procedure P11
                Déclarations de P11

                begin (* début de P11 *)
                corps de P11
                end;

                procedure P12
                Déclarations de P12

                begin (* début de P12 *)
                corps de P12
                end;

        begin (* début de P1 *)
        corps de P1
        end;

        procedure P2
        Déclarations de P2

        begin (* début de P2 *)
        corps de P2
        end;

        procedure P3
        Déclarations de P3
        ... ...
        begin (* début de P3 *)
        corps de P3
        end;

begin (* début du programme principal *)
corps du programme principal
end.
```

P3 peut appeler P1 et P2; P12 peut appeler P11, mais, dans l'état actuel, P1 ne peut appeler ni P2 ni P3, P2 ne peut appeler P3 et P11 ne peut appeler P12.

Il est cependant possible d'annoncer l'existence d'une procédure avant d'en donner l'entête et le texte, grâce à la mention "forward". Ainsi on peut modifier le texte global du programme proposé ci-dessus :

```
Program Principal;
Déclarations du programme principal

        procedure P2; forward;
        procedure P3; forward;

        procedure P1
        Déclarations de P1


                procedure P12;forward;

                procedure P11
                Déclarations de P11

                begin
                corps de P11
                end;

                procedure P12
                Déclarations de P12

                begin
                corps de P12
                end;

        begin
        corps de P1
        end;

        procedure P2
        Déclarations de P2

        begin
        corps de P2
        end;

        procedure P3
        Déclarations de P3
        ... ...
        begin
        corps de P3
        end;

begin
corps du programme principal
end.
```

On constate qu'on y a annoncé avant P1 l'existence des procédures P2 et P3. Dès lors P1 pourra appeler P2 et P3. La même déclaration permet de faire en sorte que P2 soit aussi précédé de l'avertissement de l'existence d'une procédure P3, à venir; dès lors P2 pourra également appeler P3.

Le même phénomène se produit pour P11 et P12 : P12 pouvait déjà appeler son aînée P11; la mention `procedure P12; forward;` avertit P11 de l'existence dans la suite de P12 et donc P11 peut aussi appeler P12.

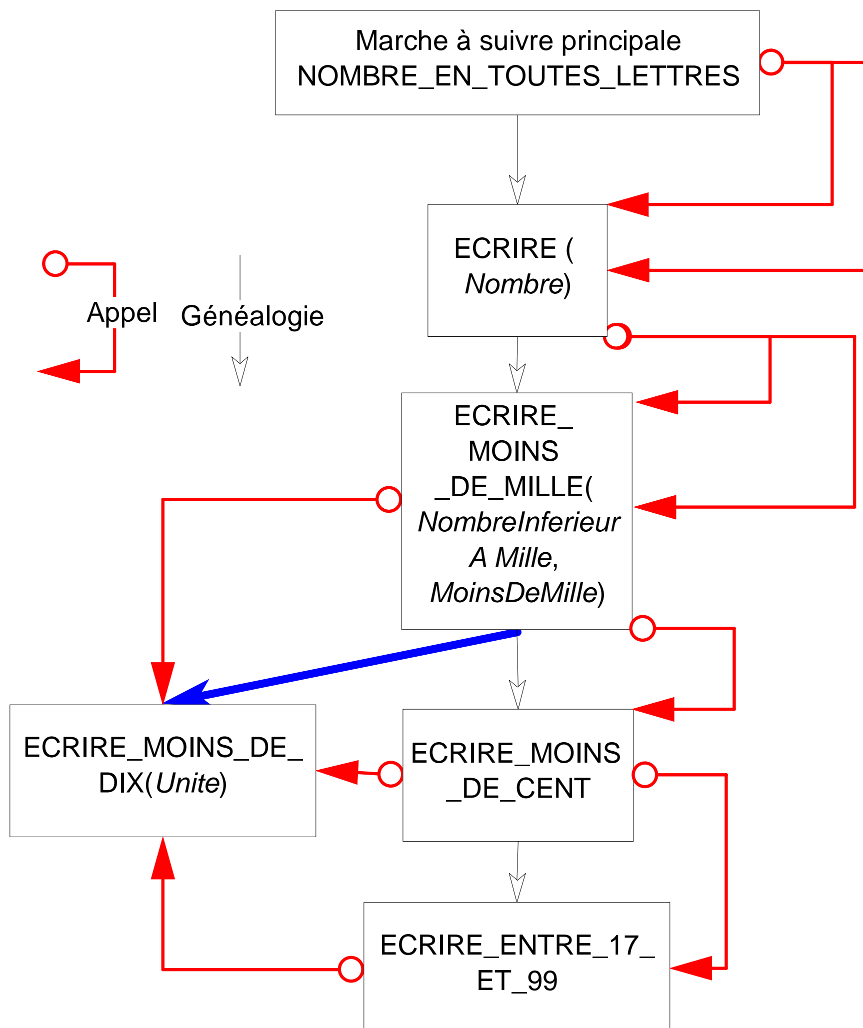
Dès lors, lorsqu'on dispose d'un ensemble de procédures soeurs, dont les textes (entêtes compris) apparaissent dans un certain ordre, il suffit de faire précéder cet ensemble des entêtes de toutes les procédures concernées, assortis de la mention "forward" pour que n'importe laquelle des procédures puisse appeler n'importe laquelle de ses soeurs, aînée ou pas.

Une dernière remarque, si les procédures sont assorties de paramètres, cette liste de paramètres doit accompagner l'entête assorti de la mention "forward" (pour permettre des appels

corrects) et lorsque l'entête véritable (celui, sans la mention "forward", qui accompagne les déclarations et le texte de la procédure) se présente, cette liste est facultative.

6. L'écriture des nombres en toutes lettres : retour sur l'organisation globale

Avec ces informations, on peut reprendre le schéma décrit page 193 et proposer la généalogie suivante :



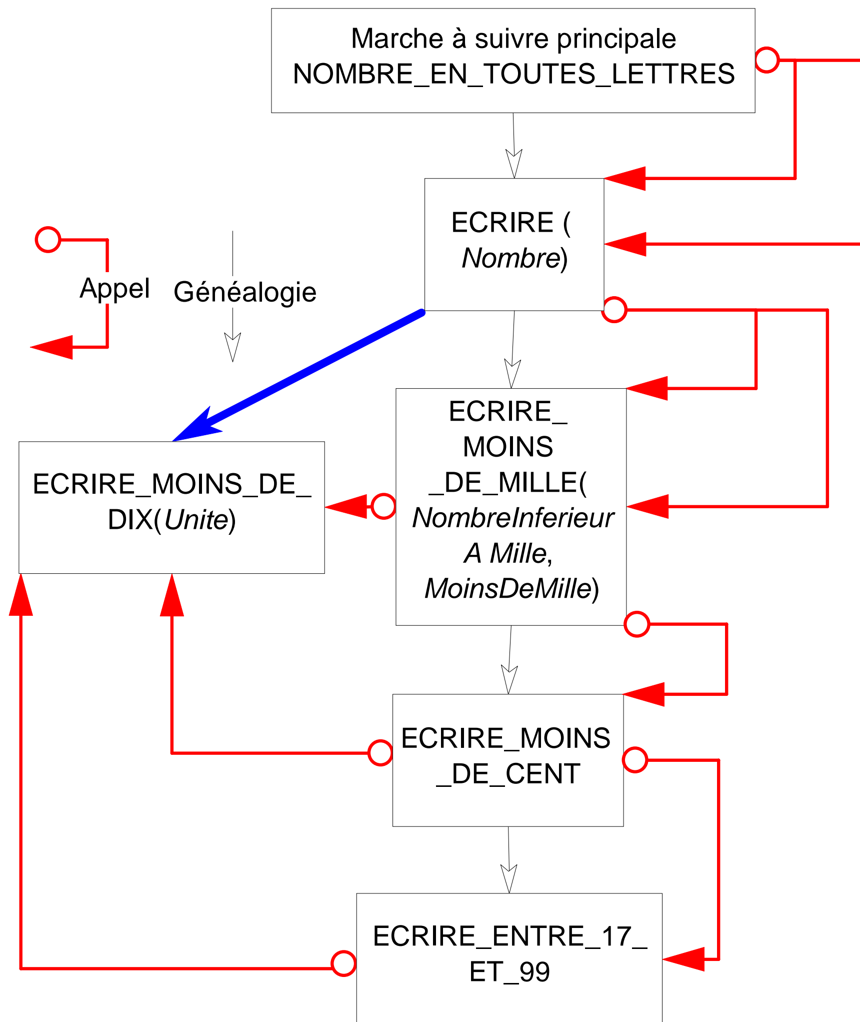
ECRIRE_MOINS_DE_DIX y est définie comme fille de ECRIRE_MOINS_DE_MILLE, soeur aînée de ECRIRE_ MOINS_DE_CENT.

Dès lors, tous les appels nécessaires :

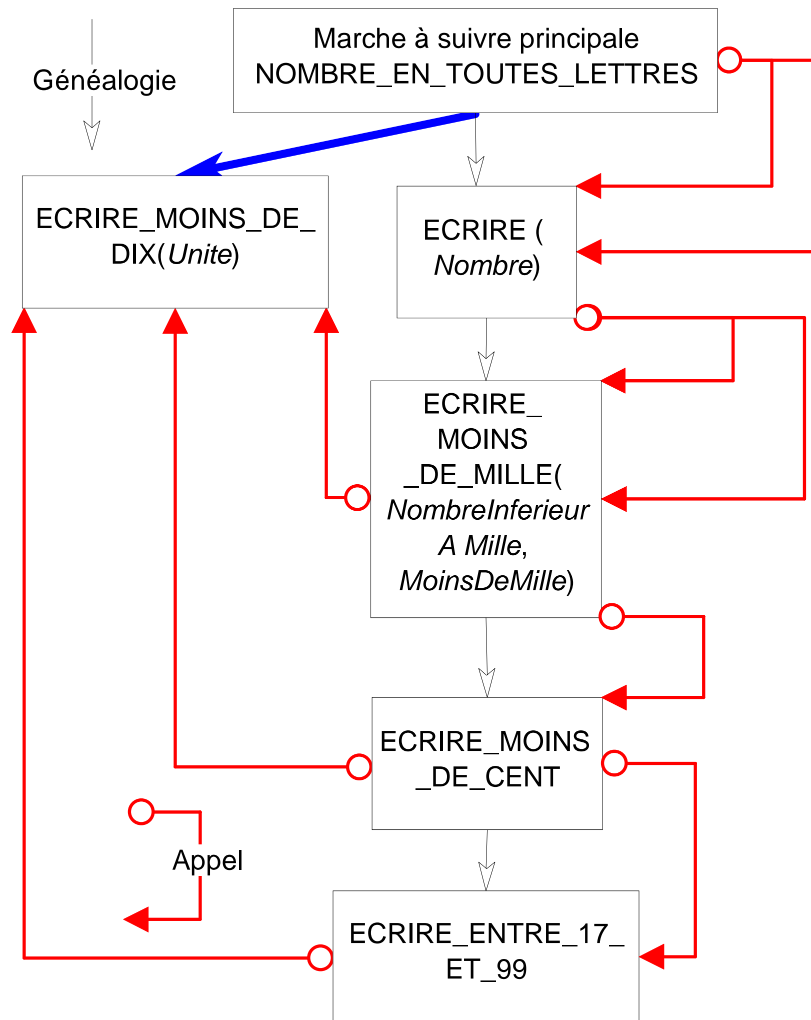
ECRIRE_MOINS_DE_MILLE	→	ECRIRE_MOINS_DE_DIX
de type mère	→	fille
ECRIRE_MOINS_DE_CENT	→	ECRIRE_MOINS_DE_DIX
de type soeur	→	soeur aînée
ECRIRE_ENTRE_17_ET_99	→	ECRIRE_MOINS_DE_DIX
de type nièce	→	tante (soeur aînée de la mère)

sont possibles.

D'autres solutions sont bien entendu envisageables, comme :



ou encore :



Dans ce dernier cas, les appels sont du type :

	ECRIRE_MOINS_DE_MILLE	→	ECRIRE_MOINS_DE_DIX
de type	nièce	→	tante (soeur aînée de la mère)
	ECRIRE_MOINS_DE_CENT	→	ECRIRE_MOINS_DE_DIX
de type	petite nièce	→	soeur aînée de la grand-mère
	ECRIRE_ENTRE_17_ET_99	→	ECRIRE_MOINS_DE_DIX
de type	arrière petite nièce	→	soeur aînée de l'arrière grand mère

Par contre d'autres solutions, comme celle où ECRIRE_ MOINS_DE_DIX serait fille de ECRIRE_MOINS_DE_CENT et soeur aînée de ECRIRE_ENTRE_17_ET_99 ne sont pas possibles. En effet, dans ce cas, ECRIRE_MOINS_DE_MILLE ne pourrait appeler sa petite-fille ECRIRE_MOINS_DE_DIX.

7. Exercices

7.1 Un peu de syntaxe

Les programmes suivants sont-ils corrects ? Sinon, pourquoi ? Si oui, quelles seront les valeurs affichées[18] ?

1

```
program P1;
var A,B,C,D :byte;

procedure PLUS_UN(I:byte;var J:byte);
var D: byte;
begin
I:=I+1;J:=J+1;C:=C+1;D:=D+1;
end;

begin
A:=0;B:=0;C:=0;D:=0;
PLUS_UN(A,B);
write(A:3,B:3,C:3,D:3);
readln;
end.
```

2

```
program P2;
var A,B,C,D :byte;

procedure PLUS_UN(I:byte;var J:byte);
var D: byte;
begin
I:=I+1;J:=J+1;A:=A+1;B:=B+1;
end;

begin
A:=0;B:=0;C:=0;D:=0;
PLUS_UN(A,B);
write(A:3,B:3,C:3,D:3);
readln;
end.
```

3

```
program P3;
var A,B,C,D :byte;

procedure PLUS_UN(I:byte;var J:byte);
var C : byte;
begin
I:=I+1;J:=J+1;A:=A+1;B:=B+1;
end;

procedure PLUS_DIX(K:byte;var L:byte);
```

18 Rappelons au passage qu'à côté du type `integer`, permettant en Pascal de coder des entiers compris entre -32768 et 32767, on trouve dans certaines implémentations (donc celle illustrée ici) les types `shortint` (permettant de coder les entiers entre -128 et 127), `longint` (entre -2147483648 et 2147483647), `byte` (entre 0 et 255) et `word` (entre 0 et 65535).

```
var D: byte;
begin
PLUS_UN(L,D);
I:=I+10;J:=J+10;K:=K+10;L:=L+10;A:=A+10;B:=B+10;C:=C+10;
D:=D+10;
end;

begin
A:=0;B:=0;C:=0;D:=0;
PLUS_DIX(A,B);
write(A:3,B:3,C:3,D:3);
readln;
end.
```

4

```
program P4;
var A,B,C,D :byte;

procedure PLUS_DIX(K:byte;var L:byte);
var D: byte;


procedure PLUS_UN(I:byte;var J:byte);
var C : byte;
begin {début de PLUS_UN }
I:=I+1;J:=J+1;A:=A+1;B:=B+1;
end;

begin  {début de PLUS_DIX }
PLUS_UN(L,D);
K:=K+10;L:=L+10;A:=A+10;B:=B+10;C:=C+10;D:=D+10;
end;

begin
A:=0;B:=0;C:=0;D:=0;
PLUS_DIX(A,B);
write(A:3,B:3,C:3,D:3);
readln;
end.
```

5

```
program P5;
var A,B,C,D :byte;

procedure PLUS_UN(I:byte;var J:byte);
var C : byte;
begin {début de PLUS_UN }
C:=C+1;D:=D+1;I:=I+1;J:=J+1;
end;

procedure PLUS_DIX(K:byte;var L:byte);
var D: byte;
begin  {début de PLUS_DIX }
PLUS_UN(K,L);
A:=A+10;B:=B+10;C:=C+10;D:=D+10;K:=K+10;L:=L+10;
end;

begin
```

```
A:=0;B:=0;C:=0;D:=0;
PLUS_DIX(A,B);
PLUS_UN(C,D);
write(A:3,B:3,C:3,D:3);
readln;
end.
```

7.2 Retour sur les chiffres et les lettres

L'écriture de la partie décimale du nombre à traiter s'effectue dans la solution proposée ci-dessus par appel de la procédure ECRIRE.

Comme cette partie décimale, exprimée en millièmes, est toujours un nombre compris entre 1 et 999, pourriez-vous proposer une solution alternative où l'écriture de cette partie décimale se ferait par appel de la procédure ECRIRE_MOINS_DE_MILLE.

7.3 Encore les chiffres et les lettres

Pourriez-vous réécrire la procédure LIRE_ET_EVALUER pour qu'elle lise un nombre comportant une partie entière, dont on préciserait par ailleurs la longueur et une partie décimale dont la longueur maximale serait aussi précisée.

7.4 Jules, des chiffres et des lettres

Pourriez-vous écrire un programme qui ferait lire un nombre en chiffres romains (entre I et MMMCMXCIX) et l'écrirait en toutes lettres (en français et pas en latin...).

7.5 Des lettres et des chiffres

Après avoir lu un chapitre suivant, traitant des chaînes de caractères, pourriez vous écrire un programme qui fasse lire un nombre (entre "un" et "neuf cent nonante neuf mille neuf cent nonante neuf" et l'affiche en chiffres (1, 2, ..., 999999) ?

8. Annexe : les chaînes de caractères.

Voici, sous forme d'un nouvel épisode "tupperware", quelques éléments à propos des chaînes de caractères de des outils qui permettent de les manipuler.

Les variables de type chaînes de caractères (string) sont en réalité des tableaux de caractères, de longueur précisée entre [ ] derrière le mot string. La toute première composante ([0]) comporte le nombre de composantes réellement occupées par les caractères composant la chaîne logée dans ce tableau :

Si un programme comporte la mention :

```
var Chaine : string[40]
```

on a en quelque sorte défini

```
var Chaine : array[0..40] of char;
```

Si on écrit ensuite

```
Chaine := 'BONJOUR'
```

la composante 0 de *Chaine* comporte alors l'octet interprétable comme l'entier 7; les composantes suivantes comportent les octets interprétables comme les caractères B, O, N,... à travers le code ASCII (ou ANSI sous Windows).

On pourra donc écrire

```
Chaine[4] := Chaine[5]
```

qui laissera inchangée la composante 0 (la longueur réellement occupée par la chaîne reste 7), mais transformera le contenu en BONOOUR.

Toute référence à une composante supérieure à 7 provoquera une erreur. Ainsi,

```
Chaine[8] := ...
```

8.1 En bref :

Une variable de type chaîne (STRING) est une séquence de caractères de longueur variable au cours de l'exécution et d'une taille prédéfinie entre 1 et 255.

La syntaxe de déclaration est :

```
   string [ constant ]
```

ou

```
   string
```

Si la longueur n'est pas spécifiée, 255 caractères sont réservés.

Les constantes de type chaîne se délimitent par des apostrophes:

```
'Turbo'
'c''est parti'
```

Remarquez comment un apostrophe est indiqué sous forme de deux apostrophes pour éviter de fermer la chaîne trop tôt.

Les opérateurs suivants peuvent être appliqués aux chaînes :

+ = <> < > <= >=

La fonction standard Length renvoie la longueur actuelle de la chaîne.

Exemples de définitions de chaînes :

```
   CONST
     LngLigne = 79;
   TYPE
     Nom   = STRING[25];
     Ligne = STRING[LngLigne];
```

8.2 Les fonctions relatives aux chaînes de caractères

8.2.1 length

length(information de type chaîne) : de type entier, désigne la longueur de la chaîne passée en argument

```
Length             Fonction
Renvoie la longueur effective d'une chaîne.
DECLARATION:
function Length(s: string): Integer;
```

8.2.2 copy

copy(information de type chaîne, information de type entier, information de type entier) : de type chaîne

Copy(chaîne dont on veut extraire une sous-chaîne,

numéro du caractère à partir duquel on extrait,

longueur de la sous-chaîne à extraire)

désigne la sous-chaîne extraite dans la chaîne mentionnée, à partir de la position indiquée et comportant le nombre de caractères indiqués)

Exemple :

copy('Crocodile', 5, 5) désigne 'odile'.

```
Copy                  Fonction
Renvoie une partie d'une chaîne de caractères.
DECLARATION:
function Copy(S: string; Position: Integer;
              Longueur: Integer): string;
DESCRIPTION: S est une expression de type chaîne. Position et
Longueur sont des expressions de type entier. Copy renvoie une
chaîne constituée de Longueur caractères, à partir du caractère
Position de la chaîne S. Si Position est supérieure à la longueur
de S, Copy renvoie une chaîne vide. Si Longueur est supérieure au
nombre de caractères restant dans la chaîne S à partir de
Position, seuls les caractères présents dans la chaîne sont
renvoyés.
```

8.2.3 concat

concat(information de type chaîne, information de type chaîne, information de type chaîne,...)

Concat désigne la chaîne obtenue par "recollement" (concaténation des diverses sous-chaînes.

On peut également noter la chaîne obtenue par concaténation à l'aide du symbole + : information de type chaîne + information de type chaîne + ...

```
Concat                Fonction
Concaténation de plusieurs chaînes de caractères.
DECLARATION:
function Concat(s1 [, s2,..., sn]: string): string;
DESCRIPTION: Chaque paramètre est une expression de type chaîne.
La chaîne résultante est la concaténation de toutes les chaînes
transmises en paramètres. Sa taille maximale est de 255
caractères. L'opérateur + donne le même résultat que la fonction
Concat:
 S := 'ABC' + 'DEF';
```

8.2.4 pos

pos(information de type chaîne, information de type chaîne) : de type integer

pos(sous-chaîne dont on teste la présence,
chaîne à explorer)

donne la position de la première occurrence éventuelle de la sous-chaîne dans la chaîne à explorer. Si la sous-chaîne n'est pas présente, on obtient 0.

Exemples :

pos('bra', 'abracadabra') désigne 2

pos('BRA', 'abracadabra') désigne 0

```
Pos                 Fonction
Recherche l'occurrence d'une sous-chaîne dans une chaîne.
DECLARATION:
function Pos(SSChaine: string; Chaine: string): Byte;
DESCRIPTION: SSChaine et Chaine sont deux expressions de type
chaîne de caractères. Pos recherche SSChaine dans Chaine et
renvoie une valeur entière représentant la position du premier
caractère de SSChaine dans Chaine. Si Pos ne trouve pas SSChaine,
elle renvoie zéro.
```

8.3 Les procédures relatives aux chaînes de caractères

8.3.1 delete

delete(*variable* de type chaîne, information de type entier, information de type entier)

delete(*variable* contenant la chaîne à raccourcir,

numéro du premier caractère de la partie à ôter,

longueur de la chaîne à ôter)

Après appel de cette procédure, la chaîne contenue dans la variable indiquée se voit ôter la sous-chaîne commençant à l'endroit mentionné et de longueur indiquée.

Exemple :

Chaine := 'Crocodile';

delete(*Chaine*, 4,2);

writeln(*Chaine*) produit alors l'affichage de Crodile

```
Delete              Procédure
Supprime une partie de chaîne.
DECLARATION:
procedure Delete(var S: string; Position: Integer; Nombre:
Integer);
DESCRIPTION: S est une variable de type chaîne. Position et Nombre
sont des expressions de type entier. Delete supprime Nombre
caractères dans la chaîne S en commençant à l'emplacement
Position. Si Position est supérieure à la longueur de S, aucun
caractère n'est supprimé. Si Nombre est supérieure au nombre de
caractères restant dans la chaîne S à partir de Position, seule la
fin de la chaîne est effacée.
```

8.3.2 insert

insert(information de type chaîne, *variable* de type chaîne, information de type entier)

Insert(sous-chaîne à insérer,

 © CeFIS - FUNDP 2002

variable contenant la chaîne où insérer la sous-chaîne,

position à partir de laquelle insérer)

Insert grossit la chaîne contenue dans la variable indiquée en lui adjoignant la sous-chaîne mentionnée à partir de la position précisée.

Exemple :

Chaine := 'Crocodile';

insert('beau', *Chaine*, 4)

writeln(*Chaine*) produit alors l'affichage de Crobeaucodile'

```
Insert              Procédure
Insère une sous-chaîne dans une chaîne.
DECLARATION:
procedure Insert(Source: string; var S: string; Pos: Integer);
DESCRIPTION: Source est une expression de type chaîne de
caractères. S est une variable de type chaîne de caractères (de
n'importe quelle longueur), et Pos une expression de type entier.
Insert insère Source dans S en position Pos. Si la chaîne
résultante a plus de 255 caractères, elle est tronquée après le
255ème caractère.
```

8.3.3 str

str(information de type entier ou réel, variable de type chaîne)

str(donnée de type entier ou réel,

variable de type chaîne)

Après l'appel, on trouve dans la variable mentionnée la chaîne de caractères correspondant au nombre fourni.

On peut aussi préciser le nombre de caractères que comportera la chaîne obtenue par conversion du nombre, les caractères constituant ce dernier étant cadré à droite :

str(information de type entier ou réel : constante entière, variable de type chaîne)

On peut aussi, préciser le nombre de décimales retenues dans la conversion du nombre :

str(information de type réel : constante entière : constante entière, variable de type chaîne)

```
Str      Procédure
Convertit une valeur numérique en une chaîne de caractères.
DECLARATION:
procedure Str(x [: width [: decimals ]]; var s: string);
DESCRIPTION: Convertit le numérique x en sa représentation chaîne
telle qu'elle serait affichée par Write.
```

8.3.4 val

val(information de type chaîne, *variable* de type entier ou réel, variable de type entier)

val(chaîne à transformer en nombre entier ou réel,

variable entière ou réelle pour accueillir le résultat,

variable contenant le numéro éventuel du premier caractère incompatible

avec l'écriture d'un entier ou d'un réel)

Après l'appel de val, la variable entière ou réelle contient le nombre correspondant à la chaîne mentionnée, si la transformation a du sens; la variable dénotant l'erreur contient alors 0.

Dans le cas où la chaîne n'est pas l'écriture convenable d'un entier ou d'un réel, la variable d'erreur comporte le numéro du premier caractère erroné.

Exemple :

Après l'appel val('123.4', *Resultat*, *Code*), la variable *Resultat* contient le réel 123.4 si *Resultat* était déclarée real; *Code* contient alors 0.

Après l'appel val('123.4', *Resultat*, *Code*), la variable *Resultat* contient le réel 123.4 si *Resultat* était déclarée integer; *Code* contient alors 4, position du point erroné dans l'écriture d'un entier.

```
Val      Procédure
Convertit une chaîne de caractères s représentant une quantité en
une valeur numérique.
DECLARATION:
procedure Val(s: string; var v; var code: Integer);
   où:
   s     =  variable chaîne; doit être une séquence
            de caractères représentant un entier.
   v     =  variable Integer ou Real
   code  =  variable Integer.
DESCRIPTION: L'action équivaut à une lecture par Read dans un
fichier texte.
```

4

Exercices

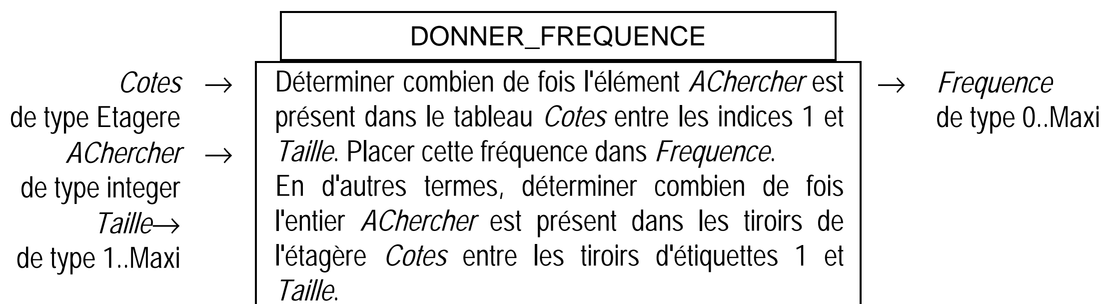
1. Analyse d'un module au sein d'un programme

1.1 Fréquence d'un élément dans un tableau

Vous participez à l'analyse d'un problème et à l'écriture du programme correspondant. Votre part du travail consiste à analyser l'action complexe DONNER_FREQUENCE qui est spécifiée comme suit :

Au premier niveau de l'analyse, sont définis :

- une constante entière Maxi = 100
- un type Etagere = array[1..Maxi] of integer



avec

Avant	Après
- *Cotes :* tableau de type Etagere - *AChercher :* contenant l'entier dont il faut dire combien de fois il est présent dans *Cotes*, dans les tiroirs entre le premier et celui portant le numéro *Taille* - *Taille :* entier entre 1 et Maxi. C'est entre les étiquettes 1 et Taille qu'il faut chercher combien de fois *AChercher* est présent dans le tableau *Cotes*	- *Cotes, AChercher* et *Taille* ne peuvent avoir changé - *Frequence* contient le nombre de fois que *AChercher* a été trouvé dans le tableau *Cotes* entre les indices 1 et *Taille*.

Pouvez-vous, en suivant le schéma utilisé ici :

- analyser l'action DONNER_FREQUENCE
- écrire la procédure Pascal correspondante.

Veillez à expliquer aussi clairement que possible votre démarche et à commenter le texte de la procédure Pascal.

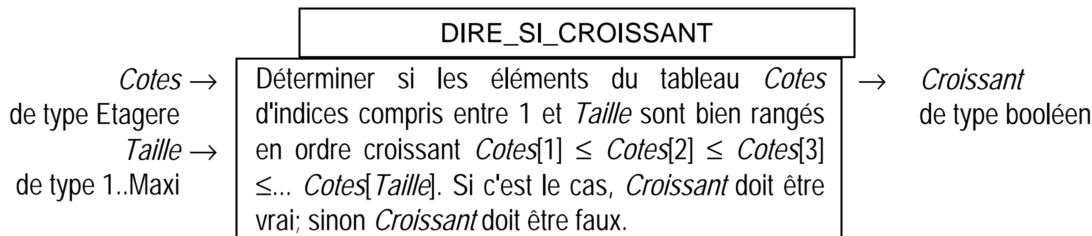
1.2 Déterminer si les éléments d'un tableau sont en ordre croissant

Vous participez à l'analyse d'un problème et à l'écriture du programme correspondant. Votre part du travail consiste à analyser l'action complexe DIRE_SI_CROISSANT qui est spécifiée comme suit :

Au premier niveau de l'analyse, sont définis :

- une constante entière Maxi = 100
- un type Etagere = array[1..Maxi] of integer

Les spécifications de l'action que vous devez analyser sont les suivantes :



avec

Avant	Après
- *Cotes :* tableau de type Etagere - *Taille :* entier entre 1 et Maxi. C'est entre les étiquettes 1 et *Taille* qu'il faut vérifier si les éléments du tableau *Cotes* sont en ordre croissant	- *Cotes* et *Taille* ne peuvent avoir changé - *Croissant* est vrai si les élements de *Cotes* d'indices entre 1 et *Taille* sont en ordre croissant et faux sinon.

Pouvez-vous, en suivant le schéma habituel :

- analyser l'action DIRE_SI_CROISSANT
- écrire la procédure Pascal correspondante.

Veillez à expliquer aussi clairement que possible votre démarche et à commenter le texte de la procédure Pascal.

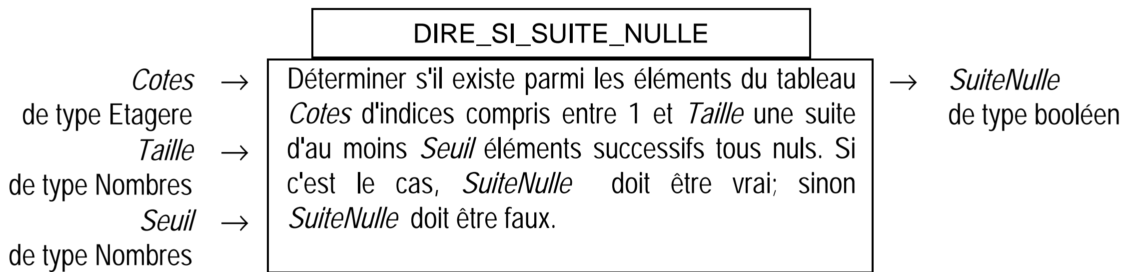
1.3 Vérifier si tous les éléments d'un tableau sont nuls

Vous participez à l'analyse d'un problème et à l'écriture du programme correspondant. Votre part du travail consiste à analyser l'action complexe DIRE_SI_SUITE_NULLE qui est spécifiée comme suit :

Au premier niveau de l'analyse, sont définis :

- une constante entière Maxi = 100
- un type énuméré Nombres = 1..Maxi;
- un type Etagere = array[Nombres] of integer;

Les spécifications de l'action que vous devez analyser sont les suivantes :



avec

Avant	Après
- *Cotes :* tableau de type Etagere - *Taille :* entier entre 1 et Maxi (de type Nombres). C'est entre les étiquettes 1 et *Taille* qu'il faut vérifier s'il existe une suite d'au moins *Seuil* éléments successifs tous nuls - *Seuil* : entier entre 1 et Maxi (de type Nombres). On cherche dans *Cotes,* entre les indices 1 et *Taille,* une suite d'au moins *Seuil* éléments successifs tous nuls.	- *Cotes, Seuil* et *Taille* ne peuvent avoir changé - *SuiteNulle* est vrai si on a pu trouver parmi les éléments de *Cotes* d'indices entre 1 et *Taille* une suite d'au moins *Seuil* éléments successifs tous nuls, et faux sinon.

Pouvez-vous, en suivant le schéma habituel :

- analyser l'action DIRE_SI_SUITE_NULLE
- écrire la procédure Pascal correspondante.

Veillez à expliquer aussi clairement que possible votre démarche et à commenter le texte de la procédure Pascal.

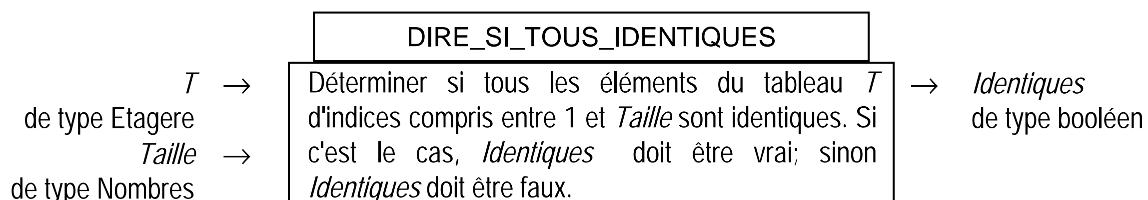
1.4 Déterminer si tous les éléments d'un tableau sont identiques

Vous participez à l'analyse d'un problème et à l'écriture du programme correspondant. Votre part du travail consiste à analyser l'action complexe DIRE_SI_TOUS_IDENTIQUES qui est spécifiée comme suit :

Au premier niveau de l'analyse, sont définis :

- une constante entière Maxi = 100
- un type énuméré Nombres = 1..Maxi;
- un type Etagere = array[Nombres] of Nombres;

Les spécifications de l'action que vous devez analyser sont les suivantes :



avec

Avant	Après
- *T :* tableau de type Etagere - *Taille :* entier entre 1 et Maxi (de type Nombres). C'est entre les étiquettes 1 et *Taille* qu'il faut vérifier que tous les éléments de *T* sont égaux	- *T* et *Taille* ne peuvent avoir changé - *Identiques* est vrai si tous les éléments de *T* d'indices entre 1 et *Taille* sont égaux, et faux sinon.

Pouvez-vous, en suivant le schéma habituel :

- analyser l'action DIRE_SI_TOUS_IDENTIQUES
- écrire la procédure Pascal correspondante.

Veillez à expliquer aussi clairement que possible votre démarche et à commenter le texte de la procédure Pascal.

1.5 Déterminer si les éléments d'un tableau sont différents d'un élément donné

Vous participez à l'analyse d'un problème et à l'écriture du programme correspondant. Votre part du travail consiste à analyser l'action complexe DIRE_SI_DIFFERENTS qui est spécifiée comme suit :

Au premier niveau de l'analyse, sont définis :

- une constante entière Maxi = 100
- un type énuméré Nombres = 1..Maxi;
- un type Etagere = array[Nombres] of Nombres;
- des variables *T*, de type Etagere
  Taille et *Repere*, de type Nombres
  Different, de type booléen

Les spécifications de l'action que vous devez analyser sont les suivantes :

avec

Avant	Après
- *T :* tableau de type Etagere - *Taille :* entier entre 1 et Maxi (de type Nombres). C'est entre les étiquettes 1 et *Taille* qu'il faut vérifier que tous les éléments de *T* sont bien différents de l'élément de *T* d'indice *Repere* - *Repere* : de type Nombres, mais toujours compris entre 1 et *Taille*; il faut vérifier que l'élément d'indice *Repere* de *T* est bien différent de tous les éléments de *T* d'indices compris entre 1 et *Taille*; évidemment l'élément d'indice *Repere* n'est pas différent de lui-même.	- *T, Repere* et *Taille* ne peuvent avoir changé - *Different* est vrai si tous les éléments de *T* d'indices entre 1 et *Taille (sauf Repere)* sont différents de l'élément de *T* d'indice *Repere*, et faux sinon.

Pouvez-vous, en suivant le schéma habituel :

- analyser l'action DIRE_SI_DIFFERENTS
- écrire la procédure Pascal correspondante.

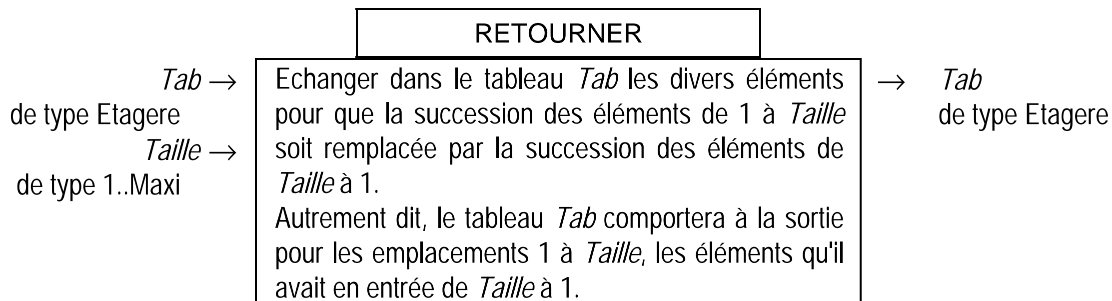
Veillez à expliquer aussi clairement que possible votre démarche et à commenter le texte de la procédure Pascal.

1.6 *Retourner un tableau*

Vous participez à l'analyse d'un problème et à l'écriture du programme correspondant. Votre part du travail consiste à analyser l'action complexe RETOURNER qui est spécifiée comme suit :

Au premier niveau de l'analyse, sont définis :

- une constante entière Maxi = 100
- un type Etagere = array[1..Maxi] of integer



avec

Avant	Après
- *Tab :* tableau de type Etagere -- *Taille :* entier entre 1 et Maxi. C'est entre les étiquettes 1 et *Taille* que le tableau doit être "retourné"	- *Tab* : les éléments de 1 à *Taille* du tableau *Tab* sont la succession des éléments de *Taille* à 1 du tableau *Tab* fourni en entrée.

Par exemple, si Maxi vaut 10 et **si Taille vaut 6**, et si *Tab* est :

2	12	-3	4	1	-4	2	-5	3	0
1	2	3	4	5	6	7	8	9	10

alors, sous l'action de RETOURNER, *Tab* doit devenir :

-4	1	4	-3	12	2	2	-5	3	0
1	2	3	4	5	6	7	8	9	10

Pouvez-vous, en suivant le schéma habituel :

- analyser l'action RETOURNER
- écrire la procédure Pascal correspondante.

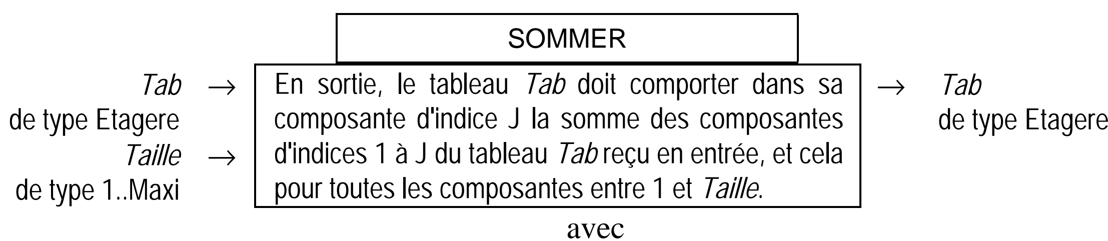
Veillez à expliquer aussi clairement que possible votre démarche et à commenter le texte de la procédure Pascal.

1.7 *Sommer les éléments d'un tableau*

Vous participez à l'analyse d'un problème et à l'écriture du programme correspondant. Votre part du travail consiste à analyser l'action complexe SOMMER qui est spécifiée comme suit :

Au premier niveau de l'analyse, sont définis :

- une constante entière Maxi = 100
- un type Etagere = array[1..Maxi] of integer



avec

Avant	Après
- *Tab :* tableau de type Etagere -- *Taille :* entier entre 1 et Maxi.	- *Tab* : la composante d'indice J du tableau *Tab* comporte la somme des composantes d'indices 1 à J du tableau *Tab* fourni en entrée, et cela pour tous les indices entre 1 et *Taille* - *Taille* n'a pas changé

Par exemple, si Maxi vaut 10 et **si Taille vaut 6**, et si *Tab* est en entrée:

2	12	-3	4	1	-4	2	-5	3	0
1	2	3	4	5	6	7	8	9	10

alors, sous l'action de SOMMER, *Tab* doit devenir :

2	14	11	15	16	12	2	-5	3	0
1	2	3	4	5	6	7	8	9	10

Pouvez-vous, en suivant le schéma habituel :

- analyser l'action SOMMER
- écrire la procédure Pascal correspondante.

Veillez à expliquer aussi clairement que possible votre démarche et à commenter le texte de la procédure Pascal.

2. Analyse de tâches

2.1 ![19]Nombre de triples lors de lancers de quatre dés

Pouvez-vous analyser le problème suivant et écrire le programme Pascal correspondant.

La tâche consiste à lancer 100 fois de suite quatre dés. On ne s'intéresse qu'au nombre de "triples" obtenus : combien de fois sur les lancers a-t-on obtenu exactement 3 "1", combien de fois exactement 3 "2", etc.

Pouvez vous mener l'analyse consistant à faire simuler cette tâche par l'ordinateur. Autant que possible :

- menez une analyse descendante
- faites clairement apparaître les constantes, types et surtout tableaux nécessaires
- écrivez un programme clair et commenté

Ce serait bien qu'avec des modifications mineures, on puisse faire plus de 100 lancers et qu'on puisse lancer à chaque fois plus de quatre dés.

[19] Les textes des programmes correspondants aux exercices marqués ! sont disponibles sur une disquette qui accompagne le présent ouvrage.

 © CeFIS - FUNDP 2002

2.2 !Nombre de piles successifs lors de lancers d'une pièce

Pouvez-vous analyser le problème suivant et écrire le programme Pascal correspondant.

La tâche consiste à lancer 1000 fois de suite une pièce de monnaie. On souhaiterait connaître à l'issue de ces lancers le nombre de fois que exactement 3 piles successifs ont été obtenus, le nombre de fois que exactement 4 piles successifs ont été obtenus et ainsi de suite jusqu'au nombre de fois où exactement 20 piles successifs ont été obtenus. Au delà de 20 piles successifs, on comptabilisera en un seul décompte toutes les séquences comptant plus de 20 piles successifs (s'il y a lieu).

Voici les sorties attendues :

```
Je vais lancer 1000 fois de suite une pièce de monnaie.
Je vous dirai ensuite combien de fois j'ai obtenu 3 "pile"
successifs, combien de fois 4 "pile" successifs, etc. jusque
combien de fois 20 piles successifs

Appuyez Entrée
```

et

```
************************  Résultats  *********************
Nombre de suites de  3 "pile" successifs : 31
Nombre de suites de  4 "pile" successifs : 8
Nombre de suites de  5 "pile" successifs : 8
Nombre de suites de  6 "pile" successifs : 9
Nombre de suites de  7 "pile" successifs : 3
Nombre de suites de  8 "pile" successifs : 0
Nombre de suites de  9 "pile" successifs : 0
Nombre de suites de 10 "pile" successifs : 0
Nombre de suites de 11 "pile" successifs : 0
Nombre de suites de 12 "pile" successifs : 0
Nombre de suites de 13 "pile" successifs : 0
Nombre de suites de 14 "pile" successifs : 0
Nombre de suites de 15 "pile" successifs : 0
Nombre de suites de 16 "pile" successifs : 0
Nombre de suites de 17 "pile" successifs : 0
Nombre de suites de 18 "pile" successifs : 0
Nombre de suites de 19 "pile" successifs : 0
Nombre de suites de 20 "pile" successifs : 0
Nombre de suites de plus de 20 "pile" successifs : 0
```

Autant que possible :

- menez une analyse descendante
- faites clairement apparaître les constantes, types et surtout tableaux nécessaires
- écrivez un programme clair et commenté

Ce serait bien qu'avec des modifications mineures, on puisse faire plus ou moins de 1000 lancers et qu'on puisse comptabiliser les suites successives de taille autre que comprise entre 3 et 20.

2.3 !Suite maximale de Pile et Face lors de lancers d'une pièce

Pouvez-vous analyser le problème suivant et écrire le programme Pascal correspondant.

La tâche consiste à lancer 1000 fois de suite une pièce de monnaie. On souhaiterait connaître à l'issue de ces lancers la taille de la plus grande suite de "pile" successifs obtenue et la taille de la plus grande suite de "face".

On veillera aussi à faire afficher à raison de 50 par ligne avec arrêt tous les 20 lignes les résultats des divers tirages en une succession de "P" ("pile") ou "F" ("face").

Voici les sorties attendues :

```
Je vais lancer 1000 fois de suite une pièce de monnaie.
Je vous dirai ensuite la taille de la plus longue suite de "pile" que
j'ai obtenue et la taille de la plus longue suite de "face" obtenue.

Appuyez Entrée
```

puis

```
FFPFPPFFPPFPPPFFFPPPPFPPPPFFFFPPFFFFFFPFPFFFFPFFPP
PFPPPFPPFFFFPPFPFFPPPPPFPFPFPFFFFFPPFPFFFPPFFPPPPF
FPPFFFPPPFFFPPFFPFPFPPPFFFPFPPFFPPFFFPFPPFFFFFPPFP
PPPFPPPPPPPPPPFFPFPFFPFPFPFPPFFPPFFFPFFPPPPFFFFPFF
FFPPFPFPPFFPFFFFFFFFFPFFFFPPPFFPFFFPPFPFPFFPFFPPFP
FPFFPPPPPPPFPFPFFPPFPFPFFPFPFFFPPPFPPPPFFFPPPPFFFF
PFFFFFFFFFFFPPPPPPFPPPFPFFFPFFFFPFFPFFPPFPPFFFPPFF
PPFFFFFPFFFFPFFFFFPFPPFFPFFFPFPFFPPFFPFPFPFFFPFFFP
PFFPFPFFPPFFPPFFPPPFFPFFPPFFPPFPPFPPPPFPFPPFFPPPFP
FPPFFFPPFFFFFFFFFPPPPPPFFFPFPPPFFPFPFPPPFFPPPPFPPP
FPPFPPFPFFFFFFPFPPPPPPPPPPFPPPFFPPPFPFPPFFFFFPPFFF
PFPFPFFPPFFPFPPFFPFFFPPPFPPPFPPFPFPFPFPPPPFPFFFPPP
PPPPPFPFPFFFFFFPFFFFFPPFPPPPPPFFPPFPPFPPFPFFFPFFFP
PFFPPPPFPFFFPFFPPFPPPPPPFPFFFPFPFPFFFPPPPPFFPPPFFF
PPPPFPFPPPPFFPFPFFFPFPPPPFPPFPFPFPPFPFPPFPPPFFFFPP
PPPFFFPPFPFPFFPFFFPFPFFFPFFFFPFFPFPFFFPPPFPPPPFPPP
PPPPFFPPFFFPFPPPPFFPPFFPPFFFPPFPFPFFFFFPPFFFFFFFPP
PPPFFFPPFFFPFFFPPFPFPFPPFFFFFFPFPFPFFFFPPPPPPFFFFP
PFPPFFFPFFFPFFPPPFPPPFPFFPFPPPPFFFFPFPFFFPFPFPFPPF
PPPFPFFPFPFFPPPPFFPPFPFPPPPPFPPPPFPPPPPFFFPPPPFPPF

************************  Résultats  ***************
Longueur de la plus grande suite de "pile"11
Longueur de la plus grande suite de "face"11
```

Autant que possible :

- menez une analyse descendante
- s'il y a lieu, faites clairement apparaître les constantes, types et surtout tableaux nécessaires
- écrivez un programme clair et commenté

Ce serait bien qu'avec des modifications mineures, on puisse faire plus ou moins de 1000 lancers.

2.4 !Différences successives lors de lancers d'un dé

Pouvez-vous analyser le problème suivant et écrire le programme Pascal correspondant.

La tâche consiste à lancer 1000 fois de suite un dé. On ne s'intéresse pas directement à la succession des résultats des lancers, mais à la valeur absolue de la différence entre ces résultats successifs. On souhaiterait connaître à l'issue de ces lancers la fréquence de ces diverses différences.

Ainsi avec 10 lancers, on aurait par exemple :

Résultats	5		2		5		6		1		4		3		1		6		1
Différences		3		3		1		5		3		1		2		5		5	

Et on annoncerait alors à la fin :

```
Fréquence de la différence 1 : 2
Fréquence de la différence 2 : 1
Fréquence de la différence 3 : 3
Fréquence de la différence 5 : 3
```

On veillera aussi à faire afficher à raison de 50 par ligne avec arrêt tous les 20 lignes les résultats des divers tirages (1, 2, 3, 4, 5 ou 6).

Autant que possible :

- menez une analyse descendante
- s'il y a lieu, faites clairement apparaître les constantes, types et surtout tableaux nécessaires
- écrivez un programme clair et commenté

Ce serait bien qu'avec des modifications mineures, on puisse faire plus ou moins de 1000 lancers.

2.5 Suites de résultats identiques lors de lancers d'un dé

Pouvez-vous analyser le problème suivant et écrire le programme Pascal correspondant.

La tâche consiste à lancer 1000 fois de suite un dé. On souhaite connaître, pour chacun des 6 résultats possibles du lancer du dé, le nombre de suites d'au moins trois résultats successifs identiques.

Ainsi avec 30 lancers, on aurait par exemple :

1 **2 2 2 2** 1 2 2 5 4 **2 2 2** 1 2 **1 1 1** 4 **3 3 3** 1 6 6 2 1 **6 6 6**

Et on annoncerait alors à la fin :

```
Nombre de suites d'au moins trois 1 : 1
Nombre de suites d'au moins trois 2 : 2
Nombre de suites d'au moins trois 3 : 1
Nombre de suites d'au moins trois 4 : 0
Nombre de suites d'au moins trois 5 : 0
Nombre de suites d'au moins trois 6 : 1
```

On veillera aussi à faire afficher à raison de 50 par ligne avec arrêt toutes les 20 lignes les résultats des divers tirages, dans l'ordre de leur apparition.

Autant que possible :

- menez une analyse descendante
- s'il y a lieu, faites clairement apparaître les constantes, types et surtout tableaux nécessaires
- écrivez un programme clair et commenté

Ce serait bien qu'avec des modifications mineures, on puisse faire plus ou moins de 1000 lancers.

2.6 !*Lancers successifs de 5 pièces*

Pouvez-vous analyser le problème suivant et écrire le programme Pascal correspondant.

La tâche consiste à lancer 1000 fois de suite cinq pièces de monnaie simultanément. On souhaite connaître, pour chacun des résultats possibles du lancer des cinq pièces, sous une forme que vous choisissez (sachant que les pièces sont indiscernables), le nombre de fois que chacun de ces résultats a été obtenu.

Autant que possible :

- faites clairement apparaître les constantes, types et surtout tableaux nécessaires
- écrivez un programme clair et commenté en choisissant des identificateurs aux noms évocateurs

Ce serait bien qu'avec des modifications mineures, on puisse faire plus ou moins de 1000 lancers.

2.7 !*Produit des résultats lors des lancers de deux dés*

Pouvez-vous analyser le problème suivant et écrire le programme Pascal correspondant.

La tâche consiste à lancer 1000 fois de suite deux dés. On s'intéresse, lors de chaque double lancer, au *produit* des deux résultats obtenus.

On souhaite que le programme à écrire

- affiche les différents produits obtenus à raison de 25 par ligne et arrêt toutes les 20 lignes (avec lecture pour reprendre), comme :

```
  6 18 16  6  4 24 12 12  4 20 10 24 15  4 12  4 10 18 30 15 30 36  4  9 20
  6  6  6 15  6 20 10  5  2  3 25  6 24 12  8 20 20  5 18 25 16  4 18  6  4
 18  2  3  4  5  9 10  4  4 24  8 10 20 18  4  6  6  6  6  4 36  3  3  3 12
 24 16  4  6 24 12  4  2 15 12  6 25 25  9 30  4  8  6 12 18  4  6  8 30 24
 15  6  5  8  6  5 24  3 16  1  4  4 25 10  5  2 12  1 12  8 16 25 18 15  9
 20  6 10 12  4 20 36  1 16 24 12  5 30  8  4 12  2 15  4 15  9  1  4  4 10
  8  5 36  6 16  4 20 12  4 20  4 12 12 12 12 12 10 36 18  6  3 15 20  6 24
  3  2 24 12  4 30 15  6 12 12 30 24  5 30 18  4  5 10 12 12 24  4  5 25  2
 15 12  3  6 36 15 36  5  8 18 24  4 18  3  3 16 16  6 12  4  8  6  1 20  6
  3 10 15  4  4 12 12  9  5 16 24 12  6 36 18 15  6  5 20  3 12 18  6  5  4
  3  2  5  5  9 16  6  5 30  6 18  8 12 15 18  3  6  4  8  9 18 12 12 18 10
 15 15  6 20 15 25 18 12 25  8  3  8 30  9  6  9  4 25 30 36 24  8 20 12 30
  4 18 36  6  9 12 16  8 25  4 18 15 24 12  9 20  6 12  4  8 12  2  2 10 10
 12  5  2  6  1 30  8  2  5 12  5 16 18 12  5 24 18 18  3 18 12 12  6  6 15
  2  5 24  8 36  6 20  4  3 36 15  3 24  4 12 36  2 20 30  1  5 10  6 12 18
  2  2  4  3  8 15  8  6 20  6  5  2 15  9 12 10  6  2  6 36  3 10 20  2 20
 10 24  5  5 15  4 12  6  2  6  3  4  6  4  5 12  8  1  4  3  5 36 12 16  2
 15  8 24  6  2 15  6 24  3 30 24 12  9 12  6 15  3  2 18  1 20  6 12 20 18
 18  6 12 10 36 24  4 16 25  2  4 12 30  3 15 10  6 24  8  8  6 25  6  6  4
  5 10 10 12  2 18 25 18 12  3 12  5 25  6  4 25  5 20  3 12 25 10 16 36 12

Appuyez Entrée pour poursuivre
```

- affiche à la fin des lancers les fréquences non nulles de chacun des produits obtenus
- affiche le nombre de fois que l'écart entre deux produits successifs a dépassé (strictement) le seuil de 20 :

 © CeFIS - FUNDP 2002

```
Appuyez Entrée pour poursuivre

Voici les fréquences des produits :
Fréquence du produit 1 =   26
Fréquence du produit 2 =   43
Fréquence du produit 3 =   57
Fréquence du produit 4 =   85
Fréquence du produit 5 =   57
Fréquence du produit 6 =  105
Fréquence du produit 8 =   61
Fréquence du produit 9 =   31
Fréquence du produit 10 =   46
Fréquence du produit 12 =  113
Fréquence du produit 15 =   56
Fréquence du produit 16 =   25
Fréquence du produit 18 =   68
Fréquence du produit 20 =   62
Fréquence du produit 24 =   59
Fréquence du produit 25 =   35
Fréquence du produit 30 =   47
Fréquence du produit 36 =   24

Le nombre de fois ou la différence entre deux produits
successifs a dépassé 20 est : 128
```

Pouvez vous mener l'analyse consistant à faire simuler cette tâche par l'ordinateur.

Autant que possible :

- menez une analyse descendante
- faites clairement apparaître les constantes, types et surtout tableaux nécessaires
- écrivez un programme clair et commenté

Ce serait bien qu'avec des modifications mineures, on puisse faire plus de 1000 lancers ou que le seuil entre deux produits successifs puisse être différent de 20.

2.8 !Lancers de 5 dés

Pouvez-vous analyser le problème suivant et écrire le programme Pascal correspondant.

La tâche consiste à lancer 1000 fois de suite cinq dés. On s'intéresse, lors de ce quintuple lancer, à **la différence entre le plus grand et le plus petit** des cinq résultats obtenus.

On souhaite que le programme à écrire

- affiche les différentes différences obtenues à raison de 25 par ligne et arrêt toutes les 20 lignes (avec lecture pour reprendre);
- affiche à la fin des lancers les fréquences non nulles de chacune des différences possibles.

Pouvez vous mener l'analyse consistant à faire simuler cette tâche par l'ordinateur.

Autant que possible :

- menez une analyse descendante,
- faites clairement apparaître les constantes, types et surtout tableaux nécessaires,
- écrivez un programme clair et commenté.

Ce serait bien qu'avec des modifications mineures, on puisse faire plus de 1000 lancers et qu'on puisse lancer plus ou moins de 5 dés à chaque fois (mais, évidemment, au moins 2)

2.9 Moyennes...

On dispose d'une liste des résultats obtenus par des élèves à un nombre variable d'interrogations. La structure de cette liste est la suivante : elle est constituée d'une suite de listes partielles comprenant successivement : le nom de lélève concerné, le nombre de résultats disponibles pour cet élève (entre 0 et 9) puis les résultats proprement dit (entiers entre 0 et 20).

Ainsi, par exemple on aura :

```
DUPONT                  nom de l'élève
3                       nombre de résultats de DUPONT
14                      les 3 résultats de DUPONT
12
9
DURAND                  nom de l'élève
0                       nombre de résultats de DURAND
DUPUIS                  nom de l'élève
1                       nombre de résultats de DUPUIS
14                      unique résultat de DUPUIS
```
etc.

La liste se termine par le nom "ZZZ" qui n'est plus suivi d'aucun résultat et ne fait évidemment pas partie des élèves.

De plus, la liste comporte à coup sûr au moins un élève et en comporte au plus 100.

On demande d'écrire un programme qui après la lecture de toutes ces données, fasse afficher la moyenne obtenue par chacun des élèves de la liste, s'il y a lieu. Ainsi, avec l'exemple donné ci-dessus, on aurait :

```
DUPONT                  nom de l'élève
11.67                   moyenne de DUPONT
DURAND                  nom de l'élève
Pas de résultat         nombre de résultats de DURANT
DUPUIS                  nom de l'élève
14                      moyenne de DUPUIS
```
etc.

On sera attentif à mener une analyse qui fasse bien apparaître les structures de données indispensables ainsi que la structure de l'algorithme créé.

On accompagnera le programme proprement dit d'un dossier expliquant la démarche et les choix effectués.

On écrira un programme Pascal lisible et commenté.

3. Syntaxe

3.1 Paramètres

Soit le programme suivant :

```
program P;
uses WinCRT;
var A,B : integer;
```

```
procedure P1(I : integer; var J : integer);
begin
I:=I+1;J:=J+1;
end;

begin
A:=0; B:=0;
P1(A,A);
P1(A,B);
writeln(A,B);
end.
```

- ce programme est-il correct ? Si oui quelles valeurs seront affichées ? sinon pourquoi ?
- que dire d'un programme parfaitement analogue, mais où l'entête de la procédure P1 serait

```
procedure P1(var I,J : integer);
```

3.2 *Paramètres*

Soit le programme suivant :

```
program P;
uses WinCRT;
var A,B : integer;

procedure P1(I : integer; J : integer);
begin
I:=I+1;J:=J+1;
end;

begin
A:=0; B:=0;
P1(A,A);
P1(B,B);
writeln(A,B);
end.
```

- ce programme est-il correct ? Si oui quelles valeurs seront affichées ? sinon pourquoi ?
- que dire d'un programme parfaitement analogue, mais où l'entête de la procédure P1 serait

```
procedure P1(var I,J : integer);
```

3.3 *Paramètres*

Soit le programme suivant :

```
program P;
uses WinCRT;
var A,B : integer;

procedure P1(I : integer; J : integer);
begin
I:=I+1;J:=J+1;
end;

begin
A:=0; B:=0;
P1(I,J);
writeln(A,B);
end.
```

- ce programme est-il correct ? Si oui quelles valeurs seront affichées ? sinon pourquoi ?
- que dire d'un programme parfaitement analogue, mais où l'entête de la procédure P1 serait

```
procedure P1(var I,J : integer);
```

3.4 Portée des variables

Soit le programme suivant :

```
program P;
var A,B : integer;

procedure P1;
var A, C : integer;
begin
A:=A+1; B:=B+1; C:=C+1;
writeln(A,B,C);
end;

begin
A:=0; B:=0; C:=0;
P1;
writeln(A,B,C);
end.
```

- ce programme est-il correct ? si oui pouvez vous dire quelles sont les six valeurs qui seront affichées ? sinon pourquoi ?
- que dire d'un programme parfaitement analogue, mais où le programme commencerait par :

```
program P;
var A, B, C : integer;
```

3.5 Paramètres

Soit le programme suivant :

```
program P;
uses WinCRT;
var A,B,C : integer;

procedure P1(J:integer);
var A, C : integer;
begin
A:=A+1; J:=J+1; C:=C+1;
end;

begin
A:=0; B:=0; C:=0;
P1(B);
writeln(A,B,C);
end.
```

- ce programme est-il correct ? Si oui quelles valeurs seront affichées ? sinon pourquoi ?
- que dire d'un programme parfaitement analogue mais où la procédure P1 commencerait par :

```
procedure P1(var J: integer);
var A : integer;
```

3.6 Erreurs

Soit le programme suivant, qui contient une procédure destinée à compter combien d'éléments figurent dans les premières composantes d'un tableau avant qu'on ne trouve 0 (comme composante) :

```
program P;
uses WinCRT;
const Maxi=100;
var T : array[1..Maxi] of integer;
```

```
    Nombre : integer;
...
      procedure COMPTER(Tab : array[1..Maxi] of integer; N: integer);
      (* COMPTER donne dans N combien de composantes non nulles on trouve dans Tab
      avant de tomber sur 0; autrement dit, à l'issue de COMPTER, N contient le
      nombre de composantes non nulles trouvées au début de Tab *)
      begin
      N:=0;
      repeat
             N:=N+1;
      until Tab[N] = 0;
      end;
...

begin
...
COMPTER(T,Nombre);
...
end.
```

- pourriez-vous mettre en évidence dans ce programme 4 erreurs importantes, en expliquant clairement pourquoi ce sont des erreurs ?
- comment corrigeriez-vous chacune des erreurs détectées (pour conduire à un programme correct) ?

3.7 !Erreurs

Soit le programme suivant, qui contient une procédure DONNER_PUISSANCE destinée à calculer un exposant :

```
program P;
uses WinCRT;
const Maxi=100;
var X,Y :integer;
    R: longint;
...
   procedure DONNER_PUISSANCE(A, B : integer; Puissance: longint);
   (* DONNER_PUISSANCE fournit dans Puissance, A^B (A exposant B), pourvu que A et
      B soient des entiers positifs ou nuls et que le résultat ne dépasse pas une
      certaine valeur maximale *)
   (* Pour rappel X^0 vaut 1 si X>0, 0^0 est indéterminé *)
   var C: integer;

   begin
   Puissance :=A;
   for C:= B-1 downto 1 do
      Puissance := Puissance * A;
   end;
...

begin
...
R:=DONNER_PUISSANCE (X,Y);
writeln (X,' exposant ', Y,' vaut ', R);
...
end.
```

- pourriez-vous mettre en évidence dans ce programme 3 erreurs importantes, en expliquant clairement pourquoi ce sont des erreurs ?
- comment corrigeriez-vous chacune des erreurs détectées (pour conduire à un programme correct) ?

3.8 Paramètres

Le programme suivant est-il correct ? Si non pourquoi ? Si oui, quelles seront les valeurs affichées :

```
program P2;
var A,B,C,D :integer;

procedure PLUS_UN(I:integer;var J:integer);
var D: integer;
begin
I := I+1;
J := J+1;
D := D+1:
end;

begin
A:=0;B:=0;C:=0;D:=0;
PLUS_UN(A,B);
write(A,B,C,D);
end.
```

4. Divers

4.1 La fonction puissance

Pascal n'offre pas d'outil pour calculer X^Y. Que pensez vous de la procédure suivante qui devrait fournir dans le paramètre variable R le résultat M^N, M et N étant des paramètres valeurs de type entier.

```
procedure PUISSANCE(M,N : byte; var R : longint);
var C : integer;
begin
R:= M;
for C:=N-1 downto 1 do
    R:=R*M;
end;
```

Ce qui est proposé est-il correct ? Si non quelles modifications apporteriez vous ?

On attend que la procédure PUISSANCE donne des résultats corrects pour $N \geq 0$ et $M \geq 0$ et M et N non tous deux nuls en même temps.

On n'est pas tenu de prévoir le cas où le résultat devient trop grand pour être contenu dans le paramètre R.

Rappel

- $M^0 = 1$, si $M \neq 0$
- $0^N = 0$, si $N \neq 0$
- 0^0 n'est pas défini

4.2 !Le reste d'une division entière

Pascal offre un outil MOD pour calculer le reste de la division entière.

 © CeFIS - FUNDP 2002

Pourriez-vous écrire une procédure DONNER_RESTE censée fournir dans un paramètre R (de type entier) le reste de la division entière de M par N, M et N étant deux paramètres de type entier. Je vous demande bien entendu, non seulement de ne pas utiliser MOD ou DIV, mais en plus de ne pas utiliser non plus * et /. En gros, les opérations sur les entiers que vous pouvez utiliser sont + et -.

DONNER_RESTE doit fournir un résultat correct pour M et N entiers strictement positifs (>0).

Bibliographie

[Armici 86 A]
ARMICI J-C.
Le Turbo Pascal version 4
Editions LI, Genève, 1986.

[Armici 86 B]
ARMICI J-C.
Programmer en Turbo-Pascal. Notions avancées.
Editions LI, Genève, 1986.

[Arsac 91]
ARSAC J.
Préceptes pour programmer
Dunod, Paris, 1991.

[Grévisse 80]
GREVISSE M.
Le bon usage (Onzième édition)
Editions Duculot, Gembloux, 1980.

[Hofstadter 87]
HOFSTADTER D., DENNETT D. (EDIT).
Vues de l'esprit.
Fantaisies et réflexions sur l'être et l'âme.
InterEditions, Paris, 1987.

[Ifrah 94]
IFRAH G.
Histoire universelle des chiffres
Bouquins, Robert Laffont, Paris, 1994

Index

—A—

adresse
- passage par ... 162

affichage des chaînes ... 31
affichage des réels ... 17
analyse de premier niveau ... 70
approche descendante ... 61
approche modulaire ... 71
au cas où... ... 137, 163

—B—

bidon (donnée) ... 11
booléen ... 36
bord
- effet de ... 162

borne (d'un tableau) ... 43
boucle For ... 49
boucle Pour ... 49
byte ... 29

—C—

calculatrice ... 117
caractère ... 34
case... of ... 136, 147, 163
César ... 117
chaînes (de caractères) ... 207
chiffres romains ... 118
chr ... 36
codage
- caractère ... 34
- entier ... 29
- réel ... 30

comp ... 30
composante (d'un tableau) ... 43
concat ... 209
constante ... 68
constante
- définition en Pascal ... 73

copy ... 209

—D—

delete ... 210
démarche descendante ... 72
donnée bidon ... 11
double ... 30

—E—

effet de bord ... 162
entier long ... 123
entrées ... 119
énuméré (type) ... 130
erreur de troncature ... 32
étagère ... 22
exécutant
- monde de l' ... 120, 166
- principal ... 149

extended ... 30

—F—

factorielle ... 52
fonction (numérique) ... 32
For... (boucle) ... 49
format d'affichage des réels ... 17
format d'affichage d'une chaîne ... 31
forward (procédure) ... 200

—G—

généalogie (des procédures) ... 192, 202
globale ... *Voir* variable

—H—

halt ... 137

—I—

indice ... 43
indicée (variable) ... 43
insert ... 210
integer ... 29
intervalle (type) ... 40

—L—

length ... 208
locale (variable) ... 85, 151
longint ... 29
Lotto (simulation du tirage) ... 62

—M—

module 70

—N—

niveau 61
niveau d'analyse 70

—O—

opération 32
ord 36, 39

—P—

paramètre 117, 140
- effectif 160
- formel 159
- utilité 197
- valeur 150
- variable 151

paramètre 125
passage par adresse 162
passage par valeur 162
poker (simulation) 93
pos 209
pour (boucle) 49
pred 36, 39
procédure
- appels possibles 194, 195, 204
- forward 200
- généalogie 202
- soeur 201

procédure soeur 194

—Q—

Quoi faire ? 119, 166

—R—

readkey (fonction) 53, 56
real 30
réels (format d'affichage) 17
règles d'écriture des nombres 167

—S—

scalaire 39, 163
shortint 29
signal 190
simulation 62
single 30
sorties 119
spécifications 119
- entrées 119
- sorties 119
- traitements 119

str211
string 207
structure de données 68
succ 36, 39

—T—

tableau 43
tâche
- monde de la 120, 121, 166

top down programming 61
traitements 119
troncature (erreur de) 32
type
- énuméré 130
- scalaire 163

type énuméré 98
type énuméré 102

—V—

val 189, 211
valeur
- passage par 162

variable
- globale 85
- locale 85
- portée 196

variable locale 151

—W—

word 29

Table des matières

AVANT PROPOS

1. DES CONSTATS 1
- **1.1 Assis entre deux chaises** 1
- **1.2 On apprend la programmation en programmant** 2
- **1.3 Le langage : quand et comment ?** 2
- **1.4 Les paramètres des procédures** 3

2. SI C'ÉTAIT À REFAIRE 4
- **2.1 Le choix des graphes de Nassi-Schneidermann (GNS)** 4
- **2.2 La métaphore de l'exécutant-ordinateur** 5
- **2.3 Les étapes du traitement informatique d'une tâche** 5

3. QUELQUES QUESTIONS IMPORTANTES 6
- **3.1 La programmation : développement ou apprentissage** 6
- **3.2 La programmation : des entrées-sorties ou des traitements** 7
- **3.3 Et l'approche objet ?** 7
- **3.4 Et dans l'enseignement secondaire ?** 8

4. ET ENFIN 8

CHAPITRE 1 : LES TABLEAUX

1. RETOUR SUR LE CALCUL DE LA MOYENNE 9
- **1.1 Énoncé : description floue** 9
- **1.2 Précisions : quoi faire (faire) ?** 9
  - 1.2.1 Une première piste sans issue… pour le moment 10
  - 1.2.2 Une solution : la donnée "bidon" 10
  - 1.2.3 Une autre solution : demander le nombre de données qui seront lues 11
  - 1.2.4 Une dernière manière de procéder 11
- **1.3 Comment faire ?** 12
- **1.4 Comment faire faire ?** 12
  - 1.4.1 Une première tentative…ratée 12
  - 1.4.2 Un replâtrage de la première tentative 14
  - 1.4.3 Une dernière proposition mieux construite 15
  - 1.4.4 Que retenir ? 16
- **1.5 Comment dire ?** 16

2. ET TOUJOURS LA MOYENNE… 19
- **2.1 Enoncé** 19
  - 2.1.1 Quoi faire ? 20
- **2.2 Comment faire retenir l'ensemble des données lues** 21
  - 2.2.1 Une solution impraticable ; des variables distinctes pour retenir les données 21
  - 2.2.2 Étagère et cie 22

2.3 Retour au problème : comment faire ? **23**
2.4 Comment faire faire ? **23**
2.4.1 Une première proposition. 23
2.4.2 Corrections 26
2.5 Comment dire **26**
3. QUELQUES COMPLÉMENTS THÉORIQUES SUR LES TYPES PRIMITIFS DE DONNÉES EN PASCAL **28**
3.1 Codage d'entiers **29**
3.1.1 Les divers types entiers 29
3.1.2 Attention aux dépassements dangereux 30
3.2 Les réels **30**
3.2.1 Codage des réels 30
3.2.2 Attention à la précision lors du codage des réels 31
3.3 Les outils de manipulation des nombres **32**
3.3.1 Les opérations 32
3.3.2 Les fonctions numériques 32
3.3.3 Opérations, fonctions, c'est finalement la même chose 33
3.4 Les caractères **34**
3.4.1 Les constantes de type caractère 34
3.4.2 Les outils de traitement des caractères 36
3.5 Le type booléen **36**
3.5.1 Les constantes booléennes 36
3.5.2 Les variables booléennes 37
3.5.3 Les outils donnant un résultat booléen et les expressions booléennes 37
3.5.4 Un petit problème 38
3.6 Les types scalaires **39**
3.7 Les types intervalles **40**
3.7.1 Des exemples 40
3.7.2 Quelques commentaires 40
4. LES TABLEAUX **41**
4.1 Le concept **41**
4.1.1 Étagères et armoires 41
4.1.2 Des exemples de déclarations 41
4.1.3 Un peu de vocabulaire 43
4.2 Les facettes essentielles du concept de tableau **43**
4.2.1 Les composantes d'un tableau donné sont toutes de même type 43
4.2.2 La grandeur d'un tableau donné est fixe et non modifiable au cours de l'exécution 43
4.2.3 Les bornes du tableau doivent être des constantes 44
4.3 L'utilisation des tableaux **44**
4.3.1 La manière de désigner une composante 44
4.3.2 Les opérations permises 44
5. ENFIN UN AUTRE EXEMPLE : AUTOUR DE JETS DE DÉS **45**
5.1 Quoi faire , **45**
5.2 Comment faire **46**
5.3 Comment faire faire **46**
5.3.1 Les variables nécessaires 46
5.3.2 Attention ! 46
5.3.3 Marche à suivre 47
5.4 Comment dire ? **48**
6. LA BOUCLE POUR... **49**

6.1 Pourquoi ? 49
6.2 La boucle Pour... "descendante" 50
6.3 Syntaxe Pascal 50
6.4 Commentaires 51
6.5 Retour sur les textes des programmes 51
6.6 Exercices 52
7. UN DERNIER EXEMPLE : LA FRÉQUENCE DES LETTRES D'UN TEXTE 52
7.1 Quoi faire 52
7.2 Une information supplémentaire 53
7.3 Retour au "quoi faire ?" 54
7.3.1 En ce qui concerne les entrées 54
7.3.2 En ce qui concerne les traitements 54
7.3.3 En ce qui concerne les sorties 54
7.4 Comment faire 54
7.5 Comment faire faire 54
7.6 Comment dire 55
8. EXERCICES SUR LES TABLEAUX 56
8.1 Manipulation 56
8.2 Utilisation 58

CHAPITRE 2 : DIVISER POUR REGNER

1. SIMULATION DU TIRAGE DU LOTTO 62
1.1 Description floue 62
1.2 Quoi faire (faire) ? 62
1.3 Comment faire ? 63
1.3.1 Première stratégie 64
1.3.2 Deuxième stratégie 65
1.4 Comment faire faire à un premier niveau d'analyse (pour la 1ère stratégie) 67
1.4.1 Structure de données. 68
1.4.2 Marche à suivre. 69
1.5 Comment dire ? (l'analyse de 1er niveau de la 1ère stratégie) 73
1.5.1 Commentaires sur le programme Pascal 73
1.6 Analyse de second (puis troisième) niveaux 74
1.6.1 Analyse de TIRER 74
1.6.2 Analyse de VERIFIER 76
1.6.3 Analyse de AFFICHER 77
1.6.4 Analyse de AVERTIR 78
1.7 L'ensemble de la solution proposée 79
1.8 Comment se passe l'exécution de l'ensemble ? 81
1.9 Comment faire faire à un premier niveau (pour la seconde stratégie) ? 86
1.9.1 Structure des données 86
1.9.2 Marche à suivre 86
1.10 Comment dire ? (l'analyse de 1er niveau de la 2ème stratégie) 89
1.11 Analyse de second niveau 90
1.11.1 Analyse de INITIALISER 90
1.11.2 Analyse de TIRER_NUMERO 90
1.11.3 Analyse de TIRER_COMPLEMENTAIRE 91
1.11.4 Analyse de AFFICHER 91
2. SIMULATION DU JEU DE POKER 93

2.1 Jeu de poker : description floue **93**
2.2 Jeu de poker : Quoi faire ? **93**
2.2.1 Les entrées 93
2.2.2 Les traitements 93
2.2.3 Les sorties 94
2.3 Jeu de poker : Comment faire ? **95**
2.4 Jeu de poker : Comment faire faire **96**
2.4.1 Structures de données 96
2.4.2 Jeu de Poker : marche à suivre de premier niveau 99
2.5 Jeu de Poker : Comment dire (1er niveau) ? **100**
2.5.1 Commentaires 102
2.6 Jeu de poker : analyse de second (et troisième) niveaux **102**
2.6.1 Analyse de FAIRE_UN_LANCER 102
2.6.2 Analyse de AFFICHER_LANCER 103
2.6.3 Analyse de EVALUER_RESULTAT 105
2.6.4 Analyse de DETERMINER... (le plus grand et le suivant dans *Lancer*) 108
2.6.5 Analyse de AVERTIR, INITIALISER_RESULTATS et AFFICHER_RESULTATS 111
3. CONCLUSIONS **112**
4. EXERCICES **113**

CHAPITRE 3 : LES LIMITES DE L'APPROCHE DESCENDANTE

1. UNE CALCULATRICE POUR LE GRAND JULES **118**
1.1 Description floue **118**
1.2 Quoi faire (faire) ? **118**
1.3 La calculatrice : Comment faire (à un premier niveau) ? **122**
1.4 La calculatrice : Comment faire faire (à un premier niveau) ? **122**
1.4.1 Structure de données. 122
1.4.2 Marche à suivre 123
1.5 La calculatrice : Comment dire (à un premier niveau) ? **127**
1.5.1 Commentaires sur le texte du programme 129
1.6 Analyse de DECIMALISER **129**
1.6.1 DECIMALISER ? Quoi faire ? 129
1.6.2 DECIMALISER ? Comment faire ? 129
1.6.3 DECIMALISER ? Comment faire faire ? 130
1.6.4 DECIMALISER. Comment dire ? 134
1.6.5 REMPLIR_VALEURS. Comment dire ? 135
1.6.6 TRANSFORMER. Comment faire faire ? 135
1.6.7 TRANSFORMER. Comment dire ? 137
1.7 Analyse de ROMANISER **138**
1.7.1 ROMANISER. Quoi faire ? 138
1.7.2 ROMANISER. Comment faire ? 138
1.7.3 ROMANISER. Comment faire faire ? 138
1.7.4 ROMANISER. Comment dire ? 141
1.8 Analyse de ROMANISER_TOUT_OU_PARTIE **142**
1.8.1 ROMANISER_TOUT_OU_PARTIE. Quoi faire ? 142
1.8.2 ROMANISER_TOUT_OU_PARTIE. Comment faire ? 142
1.8.3 ROMANISER_TOUT_OU_PARTIE. Comment faire faire ? 143
1.8.4 ROMANISER_TOUT_OU_PARTIE. Comment dire ? 146
1.9 Analyse de CALCULER_10_EXP_C_DIV_2 **147**

1.9.1 CALCULER_10_EXP_C_DIV_2. Comment dire 147
1.10 Et à l'exécution ? : les paramètres 148
1.10.1 Début du travail de l'exécutant principal et de son installateur d'étiquettes 148
1.10.2 Le premier appel de l'exécutant chargé de ROMANISER 150
1.10.3 Début de l'exécution de ROMANISER et appel de ROMANISER_TOUT_ OU_PARTIE 152
1.10.4 Le travail de ROMANISER_TOUT_OU_PARTIE et l'appel de CALCULER_10_EXP_C_DIV_2 154
1.10.5 Poursuite et fin du travail de ROMANISER_TOUT_OU_PARTIE 155
1.10.6 Fin du travail de ROMANISER 156
1.10.7 Poursuite du travail de l'exécutant principal et nouvel appel de ROMANISER 156
2. LES PARAMÈTRES D'UNE PROCÉDURE 159
2.1 Pourquoi 159
2.2 Comment 159
2.2.1 Les principes 159
2.2.2 Les détails syntaxiques en Pascal 162
3. COMPLÉMENTS SUR PASCAL 163
3.1 La structure "Case of..." 163
4. DES CHIFFRES ET DES LETTRES 165
4.1 Les nombres en toutes lettres : description floue 165
4.2 Les nombres en toutes lettres : "Quoi faire ?" 165
4.3 Les nombres en toutes lettres : "Comment faire à un premier niveau ?" 167
4.4 Les nombres en toutes lettres : "Comment faire faire à un premier niveau ?" 169
4.4.1 Structure de données 169
4.4.2 Marche à suivre 170
4.5 Les nombres en toutes lettres : "Comment dire à un premier niveau ?" 173
4.6 Analyse de AVERTIR 174
4.7 Analyse de RACCOURCIR 174
4.7.1 RACCOURCIR : Comment faire ? 174
4.7.2 RACCOURCIR : Comment faire faire ? 175
4.7.3 RACCOURCIR : Comment dire ? 175
4.8 Analyse de FAIRE_SUIVRE_DE_SA_NATURE 175
4.8.1 FAIRE_SUIVRE_DE_SA_NATURE : Comment faire ? 175
4.8.2 FAIRE_SUIVRE_DE_SA_NATURE : Comment faire faire ? 176
4.8.3 FAIRE_SUIVRE_DE_SA_NATURE : Comment dire ? 177
4.9 Analyse de ECRIRE 178
4.9.1 ECRIRE : Comment faire ? 178
4.9.2 ECRIRE : Comment faire faire ? 178
4.9.3 ECRIRE : Comment dire ? 179
4.10 Analyse de ECRIRE_MOINS_DE_MILLE 180
4.10.1 ECRIRE_MOINS_DE_MILLE : Comment faire ? 180
4.10.2 ECRIRE_MOINS_DE_MILLE : Comment faire faire ? 180
4.11 Analyse de ECRIRE_MOINS_DE_CENT 182
4.11.1 ECRIRE_MOINS_DE_CENT : Comment faire ? 182
4.12 Analyse de ECRIRE_MOINS_DE_DIX 182
4.12.1 ECRIRE_MOINS_DE_DIX : Comment faire faire ? 182
4.12.2 ECRIRE_MOINS_DE_DIX : Comment dire ? 183
4.13 Retour à l'analyse de ECRIRE_MOINS_DE_MILLE 183
4.13.1 ECRIRE_MOINS_DE_MILLE : une nouvelle version du "Comment faire faire ?" 183
4.13.2 ECRIRE_MOINS_DE_MILLE : Comment dire ? 183
4.14 Retour à l'analyse de ECRIRE_MOINS_DE_CENT 184
4.14.1 ECRIRE_MOINS_DE_CENT : Comment faire faire ? 184

4.14.2 ECRIRE_MOINS_DE_CENT : Comment dire ? 185
4.15 Analyse de ECRIRE_ENTRE_17_ET_99 185
4.15.1 ECRIRE_ENTRE_17_ET_99 : Comment faire ? 185
4.15.2 ECRIRE_ENTRE_17_ET_99 : Comment faire faire ? 186
4.15.3 ECRIRE_ENTRE_17_ET_99 : Comment dire ? 186
4.16 Analyse de LIRE_ET_EVALUER 187
4.16.1 LIRE_ET_EVALUER : Comment faire ? 187
4.16.2 LIRE_ET_EVALUER : Comment faire faire ? 188
4.16.3 LIRE_ET_EVALUER : Comment dire ? 189
4.17 Analyse de VERIFIER_ET_SCINDER 190
4.17.1 VERIFIER_ET_SCINDER : Comment faire ? 190
4.17.2 VERIFIER_ET_SCINDER : Comment faire faire ? 191
4.17.3 VERIFIER_ET_SCINDER : Comment dire ? 191
4.18 L'organisation générale du programme global 192
5. LES PROCÉDURES : QUI PEUT APPELER QUI ? 194
5.1 Les appels possibles 194
5.2 Appels de procédures et portée des variables 195
5.3 La possibilité du FORWARD 200
6. L'ÉCRITURE DES NOMBRES EN TOUTES LETTRES : RETOUR SUR L'ORGANISATION GLOBALE 202
7. EXERCICES 205
7.1 Un peu de syntaxe 205
7.2 Retour sur les chiffres et les lettres 207
7.3 Encore les chiffres et les lettres 207
7.4 Jules, des chiffres et des lettres 207
7.5 Des lettres et des chiffres 207
8. ANNEXE : LES CHAÎNES DE CARACTÈRES. 207
8.1 En bref : 208
8.2 Les fonctions relatives aux chaînes de caractères 208
8.2.1 length 208
8.2.2 copy 209
8.2.3 concat 209
8.2.4 pos 209
8.3 Les procédures relatives aux chaînes de caractères 210
8.3.1 delete 210
8.3.2 insert 210
8.3.3 str
8.3.4 val

EXERCICES

1. ANALYSE D'UN MODULE AU SEIN D'UN PROGRAMME 213
1.1 Fréquence d'un élément dans un tableau 213
1.2 Déterminer si les éléments d'un tableau sont en ordre croissant 214
1.3 Vérifier si tous les éléments d'un tableau sont nuls 214
1.4 Déterminer si tous les éléments d'un tableau sont identiques 215
1.5 Déterminer si les éléments d'un tableau sont différents d'un élément donné 216
1.6 Retourner un tableau 217
1.7 Sommer les éléments d'un tableau 217

2. ANALYSE DE TÂCHES 218
2.1 Nombre de triples lors de lancers de trois dés 218
2.2 Nombre de piles successifs lors de lancers d'une pièce 219
2.3 Suite maximale de Pile et Face lors de lancers d'une pièce 219
2.4 Différences successives lors de lancers d'un dé 220
2.5 Suites de résultats identiques lors de lancers d'un dé 221
2.6 Lancers successifs de 5 pièces 222
2.7 Produit des résultats lors des lancers de deux dés 222
2.8 Lancers de 5 dés 223
2.9 Moyennes... 224
3. SYNTAXE 224
3.1 Paramètres 224
3.2 Paramètres 225
3.3 Paramètres 225
3.4 Portée des variables 226
3.5 Paramètres 226
3.6 Erreurs 226
3.7 Erreurs 227
3.8 Paramètres 228
4. DIVERS 228
4.1 La fonction puissance 228
4.2 Le reste d'une division entière 228