# *INTRODUCTION A L'INFORMATIQUE*

## *LANGAGE PASCAL*

version 1.4

Pierre Breguet

octobre 1992

---

## Table des matières

---

---

[ Retour au sommaire des cours ]

---

Mise à jour par Liborio MOGAVERO, le 28 janvier 1997

# Table des matières

## Avant-propos

[ Retour à la page d'accueil | Suivant | Conventions d'écritures | Références | Index | Bibliographie | Annexes ]

## Conventions d'écritures

[ Retour à la page d'accueil | Suivant | Précédent | Avant-propos | Conventions d'écritures | Références | Index | Bibliographie | Annexes ]

## Références

[ Retour à la page d'accueil | Suivant | Précédent | Avant-propos | Conventions d'écritures | Références | Index | Bibliographie | Annexes ]

---



[ Retour à la page d'accueil | Suivant | Précédent | Avant-propos | Conventions d'écritures | Références | Index | Bibliographie | Annexes ]

[ Retour à la page d'accueil | Précédent | Suivant | Avant-propos | Conventions d'écritures | Références | Index | Bibliographie | Annexes ]

## Chapitre 3 - Les nombres et les entrées-sorties de bases

[ Retour à la page d'accueil | Précédent | Suivant | Avant-propos | Conventions d'écritures | Références | Index | Bibliographie | Annexes ]

## Chapitre 4 - L'itération, le choix, les autres types standard

[ Retour à la page d'accueil | Précédent | Suivant | Avant-propos | Conventions d'écritures | Références | Index | Bibliographie | Annexes ]

## **Chapitre 5 -** Les procédures et les fonctions

[ Retour à la page d'accueil | Précédent | Suivant | Avant-propos | Conventions d'écritures | Références | Index | Bibliographie | Annexes ]

## Chapitre 6 - Les tableaux et les enregistrements



[ Retour à la page d'accueil | Précédent | Suivant | Avant-propos | Conventions d'écritures | Références | Index | Bibliographie | Annexes ]

## Chapitre 7 - Les types enumérés et intervalle

[ Retour à la page d'accueil | Précédent | Suivant | Avant-propos | Conventions d'écritures | Références | Index | Bibliographie | Annexes ]

## Chapitre 8 - Les ensembles et les chaînes de caractères

[ Retour à la page d'accueil | Précédent | Suivant | Avant-propos | Conventions d'écritures | Références | Index | Bibliographie | Annexes ]

## Chapitre 9 - Les types pointeur



[ Retour à la page d'accueil | Précédent | Suivant | Avant-propos | Conventions d'écritures | Références | Index | Bibliographie | Annexes ]

## Chapitre 10 - Les fichiers

[ Retour à la page d'accueil | Précédent | Suivant | Avant-propos | Conventions d'écritures | Références | Index | Bibliographie | Annexes ]

## Chapitre 11 - La récursivité



[ Retour à la page d'accueil | Précédent | Suivant | Avant-propos | Conventions d'écritures | Références | Index | Bibliographie | Annexes ]

## Chapitre 12 - Compléments de Pascal

[ Retour à la page d'accueil | Précédent | Suivant | Avant-propos | Conventions d'écritures | Références | Index | Bibliographie | Annexes ]

---

## Index

[ Retour à la page d'accueil | Précédent | Suivant | Avant-propos | Conventions d'écritures | Références | Bibliographie | Annexes ]

## Bibliographie

[ Retour à la page d'accueil | Précédent | Suivant | Avant-propos | Conventions d'écritures | Références | Index | Annexes ]

---

# Annexes

---

---

[ Retour à la page d'accueil | Précédent | Avant-propos | Conventions d'écritures | Références | Index | Bibliographie ]

# AVANT-PROPOS

Ce cours est principalement destiné aux étudiants de première année de la division d'électricité de l'Ecole d'Ingénieurs de l'Etat de Vaud. Il a été rédigé en tenant compte d'une approche pédagogique et non en vue de l'établissement d'un manuel de référence d'un dialecte du langage Pascal. Cela signifie pas que la précision et la rigueur ont été négligées. Au contraire toutes les notions ont été présentées sous l'angle dit "de la simplicité dans la précision", du moins l'auteur espère-t-il !

Mentionnons également le soin apporté par le professeur Jean-Pierre Molliet à la lecture et à la critique de ce document Qu'il en soit vivement remercié ! Merci à lui pour son support de cours dont s'inspire le chapitre 11.

[ Retour à la table des matières ]

# CONVENTIONS D'ECRITURES

Ce cours utilise les conventions suivantes :

- tout mot réservé du langage Pascal est écrit en **caractères gras**

- tout identificateur prédéfini du langage Pascal est écrit en *italiques*.

- tout terme français important devant être mémorisé est en **gras**

- toute partie de programme reprise dans le texte ou dans les commentaires du programme est écrite en *italiques* .

[ Retour à la table des matières ]

# REFERENCES

Les différents dialectes du langage Pascal utilisés sont les suivants :

- Pascal Macintosh, nom abrégé pour THINK's Lightspeed Pascal version 3.0
- Pascal VAX, nom abrégé pour VAX Pascal version 4.0
- Turbo Pascal, nom abrégé pour Turbo Pascal version 6.0

[ Retour à la table des matières ]

**CHAPITRE 1**

# INTRODUCTION

## 1.1 **Informatique, algorithmes et programmation**

Le mot informatique est un mot français ("computer science" en anglais) désignant la science qui traite des données pour obtenir des informations (résultats). Ce traitement est traditionnellement effectué à l'aide d'algorithmes.

Un **algorithme** est une suite d'opérations à effectuer pour résoudre un problème.

L'**algorithmique** est la partie de l'informatique qui traite des algorithmes.

**Exemples d'algorithmes**:

- Un algorithme de résolution de l'équation ax+b = 0 est:
  1. Soustraire b à gauche et à droite du signe =. On obtient ax = -b
  2. Si a est non nul diviser chaque membre par a. On obtient le résultat cherché qui est x = -b/a
  3. Si a est nul l'équation est insoluble
- Un algorithme de mise en marche d'une voiture est:
  1. Mettre la clé dans le démarreur
  2. Serrer le frein à main
  3. Mettre le levier des vitesses au point mort
  4. Répéter les opérations suivantes
     - tourner la clé dans le sens des aiguilles d'une montre
     - attendre quelques secondes
     - mettre la clé dans la position "marche"
     - si le moteur ne tourne pas, ramener la clé dans sa position initiale jusqu'à ce que le moteur démarre
  5. Enclencher la première vitesse
  6. Desserrer le frein à main

Lorsqu'un ordinateur doit suivre un algorithme pour résoudre un problème, cet algorithme doit être exprimé dans un langage compréhensible par la machine. Ce langage appelé langage machine est composé de suites de 0 et de 1.

Or le programmeur ne peut exprimer des algorithmes complexes avec des 0 et des 1! Il va utiliser un langage plus proche d'une langue naturelle et appelé **langage de programmation.**

Un fois cet algorithme codé dans un langage de programmation le programme ainsi créé doit être

- soit traduit complètement en langage machine par le **compilateur**
- soit directement **interprété**, c'est-à-dire que chaque ligne de code est exécutée directement sans une traduction de tout le programme. Cette ligne peut ou non être traduite en un langage proche du langage machine avant exécution.

La **compilation** est une étape supplémentaire mais a l'avantage de produire un programme en langage machine. Ce programme en langage machine peut exister aussi longtemps que nécessaire. Chaque fois que le programme doit être utilisé,

il le sera directement, ce qui implique que la compilation n'est nécessaire que la première fois.

L'**interprétation** est plus directe mais la traduction de chaque ligne a lieu à chaque utilisation du programme.

L'**exécution** par un ordinateur d'un programme compilé sera donc nettement plus rapide que l'exécution d'un programme par interprétation.

Enfin la **programmation** est la technique utilisée pour écrire des programmes en langage de programmation.

[ Retour à la table des matières ]

## 1.2 **Langages de programmation**

Voici par ordre chronologique les principaux langages de programmation utilisés aujourd'hui ainsi que leur principal domaine d'application:

| Année | Langage | Domaine d'application | Remarques |
|---|---|---|---|
| 1955 | Fortran | Calcul scientifique | Langage ancien, dont les versions plus récentes comportent des bizarreries héritées des années 50! |
| 1958 | Algol-60 | Algorithmique | Premier langage algorithmique |
| 1959 | Lisp | Intelligence artificielle | Premier langage non traditionnel |
| 1961 | Cobol | Gestion | Langage "verbeux", peu maniable |
| 1964 | PL/1 | Langage général | Complexe, créé et utilisé chez IBM |
| 1965 | Basic | "Travail à la maison" | Simple d'utilisation, peu adapté à la programmation structurée |
| 1970 | Prolog | Intelligence artificielle | Langage non algorithmique |
| 1971 | Pascal | Enseignement | Créé à Zurich, diffusé partout, souffre des nombreuses versions différentes d'un ordinateur à l'autre |
| 1972 | C | Programmation système | Accès facile au matériel... |
| 1980 | Modula-2 | Programmation système | Descendant direct de Pascal, mêmes problèmes de versions différentes |
| 1983 | Ada | Langage général | Complexe, puissant, langage du futur? |

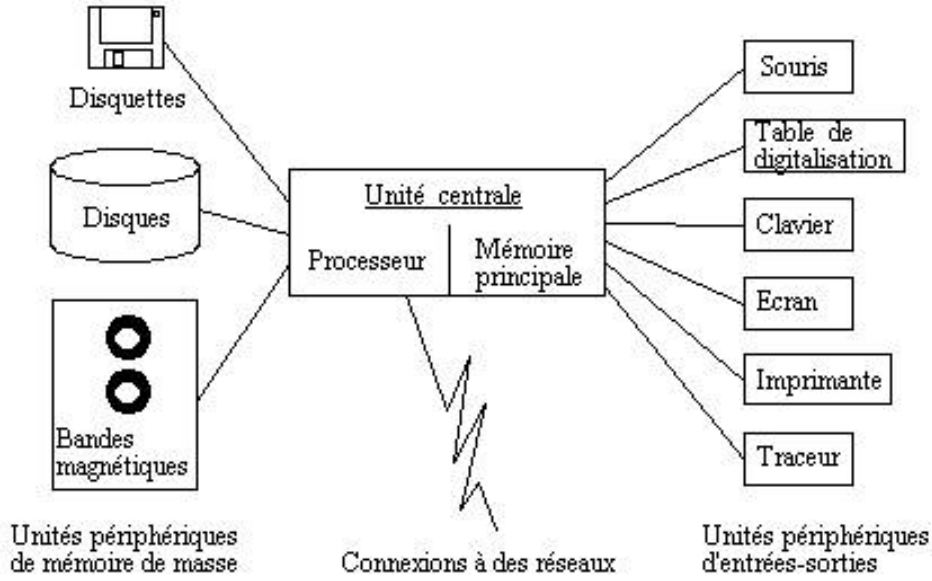
Mentionnons également les langages d'assemblage appelés assembleurs, utilisés surtout pour programmer au niveau de la machine.

[ Retour à la table des matières ]

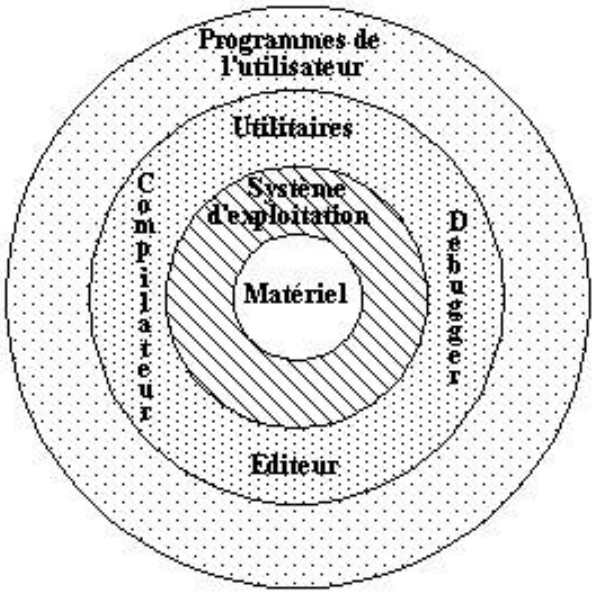
## 1.3 **Qu'est-ce qu'un ordinateur?**

Un ordinateur est une machine créée pour exécuter très rapidement et de manière très fiable des tâches complexes. C'est un assemblage de composants matériels et logiciels. Le **matériel** (hardware en anglais) comporte toutes les parties physiques alors que le **logiciel** (software en anglais) représente tous les programmes nécessaires au fonctionnement du matériel.

Le matériel peut être schématisé ainsi:




Le logiciel peut se représenter de la manière suivante:




L'apparence externe d'un ordinateur est très variable (observer un Macintosh, un Olivetti M250, un Vax, un Cray...). Tous sont cependant formés des éléments cités ci-dessus.

[ Retour à la table des matières ]

## 1.4 **Catégories d'ordinateurs**

Les ordinateurs actuels peuvent être classés en 5 catégories selon leurs domaines d'application:

a) **Supercalculateurs** (et non superordinateurs!)

Machines destinées au calcul scientifique, extrêmement rapides.

Exemples typiques: Cray-YMP, NEC

b) **Ordinateurs de moyenne puissance**

Machines destinées au calcul scientifique et aux applications de gestion.

Exemples typiques: IBM AS/400, Vax 9000, Burroughs

c) **Miniordinateurs**

Machines de développement, CAO.

Exemples typiques: DEC Vax 6000-510, Data General série MV, HP 9000

d) **Stations individuelles**

CAO, commande de processus, machines de développement.

Exemples typiques: Sun 3, Sun SPARC, Apollo

e) **Ordinateurs personnels (PC)**

Apprentissage, machines de bureau, commande de processus.

Exemples typiques: Macintosh SE/30, IBM PS/2, machines à base de 386 et 486

**Remarque importante:**

Chaque ordinateur est adapté à une (ou quelques) catégorie(s) d'applications mais aucun ne pourra convenir agréablement à plusieurs applications fondamentalement différentes.

[ Retour à la table des matières ]

## 1.5 **Le langage Pascal**

L'expérience montre que le premier langage appris est fondamental pour le futur d'un programmeur. En effet les habitudes prises sont ancrées si profondément qu'il est très difficile de les modifier voire de s'en défaire!

L'apprentissage de la programmation doit donc s'effectuer avec un langage forçant le programmeur à adopter de "bonnes habitudes" (cf. 1.6).

D'autre part il est toujours plus facile d'apprendre et d'utiliser de nouveaux langages lorsque ces "bonnes habitudes" sont acquises.

Le langage Pascal aide à l'apprentissage d'une bonne programmation en obligeant le programmeur à se soumettre à certaines contraintes et en lui fournissant une panoplie assez riche d'outils agréables à utiliser.
Ces outils vont permettre de coder relativement simplement un algorithme pas trop complexe, de refléter fidèlement sa structure.

[ Retour à la table des matières ]

## 1.6 **Bonnes habitudes de programmation**

Le but de tout programmeur est d'écrire des programmes **justes, simples, lisibles** par d'autres programmeurs, **fiables** et **efficaces**. Pour cela les quelques points suivants sont fondamentaux (liste non exhaustive!):

```
1. Réfléchir et imaginer de bons algorithmes de résolution
   avant d'écrire la première ligne du programme.

2. Une fois l'algorithme trouvé, en écrire l'esquisse en
   français, puis le coder dans le langage de programmation choisi.

3. Lors du codage:
        - choisir des noms parlants pour représenter les objets
```

manipulés dans le programme
- **commenter** chaque morceau du programme de manière explicative et non descriptive
- **tester** chaque module, procédure, fonction séparément

[ Retour à la table des matières ]

## 1.7 **De l'analyse du problème à l'exécution du programme**

Voici une marche à suivre pour la création de programmes à partir d'un problème donné.

1. Bien lire l'énoncé du problème, être certain de bien le comprendre.
2. Réfléchir au problème, déterminer les points principaux à traiter.
3. Trouver un bon algorithme de résolution (cf. 2.2), l'écrire sur papier et en français.
4. Coder l'algorithme en un programme écrit sur papier.
5. Introduire le programme dans l'ordinateur au moyen d'un éditeur.
6. Vérifier la syntaxe du programme au moyen d'un vérificateur de syntaxe.
7. Compiler (puis linker) le programme
8. Exécuter le programme, vérifier son bon fonctionnement par des tests

En cas d'erreurs de syntaxe ou d'erreurs à la compilation il faut les corriger avec l'éditeur puis reprendre au point 6.

Si le programme fonctionne mais donne des résultats faux, cela signifie qu'il y a des erreurs de logique. Il faut réfléchir, les trouver, modifier le programme en conséquence puis reprendre au point 6.

[ Retour à la table des matières ]

**CHAPITRE 2**

# EXEMPLE INTRODUCTIF ET METHODE DE DECOMPOSITION

---

## 2.1 **Exemple introductif**

Nous allons examiner un problème (simple) dont la résolution sera effectuée en respectant la marche à suivre donnée au chapitre 1 (cf. 1.7), restreinte aux points 1 à 4. Pour trouver un algorithme de résolution (points 2 et 3) nous allons appliquer une méthode qui devrait être utilisée chaque fois qu'un problème de programmation doit être résolu. Cette méthode est connue sous le nom de **méthode de décomposition par raffinements successifs.**

Nous allons également nous baser sur cet exemple pour rappeler des bonnes habitudes de programmation et pour introduire les premières notions de programmation en langage Pascal.

L'exemple est le suivant:

> Dessiner sur l'écran du Macintosh une figure composée de deux formes géométriques
> (carré, triangle isocèle), chacune comportant son nom en forme de titre.

Sans être trop optimiste la compréhension de ce problème est immédiate, après avoir précisé que la disposition des formes est libre et qu'un titre doit se situer au-dessus de "sa" forme (point 1 résolu).

[ Retour à la table des matières ]

## 2.2 **Méthode de décomposition par raffinements successifs**

Cette méthode est basée sur l'idée suivante:

Etant donné un problème, le décomposer en sous-problèmes de telle manière que

- chaque sous-problème soit une partie du problème donné
- chaque sous-problème soit plus simple (à résoudre) que le problème donné
- la réunion de tous les sous-problèmes soit équivalente au problème donné

puis reprendre chaque sous-problème et le décomposer comme ci-dessus. Une étape de cette suite de décompositions est appelée **raffinement**.

Cette méthode est efficace après l'avoir utilisée plusieurs fois. Elle est indispensable lorsque les problèmes deviennent complexes. Cependant son application dans des cas simples donne de bons algorithmes de résolution des problèmes.

[ Retour à la table des matières ]

## 2.3 **Application à l'exemple introductif**

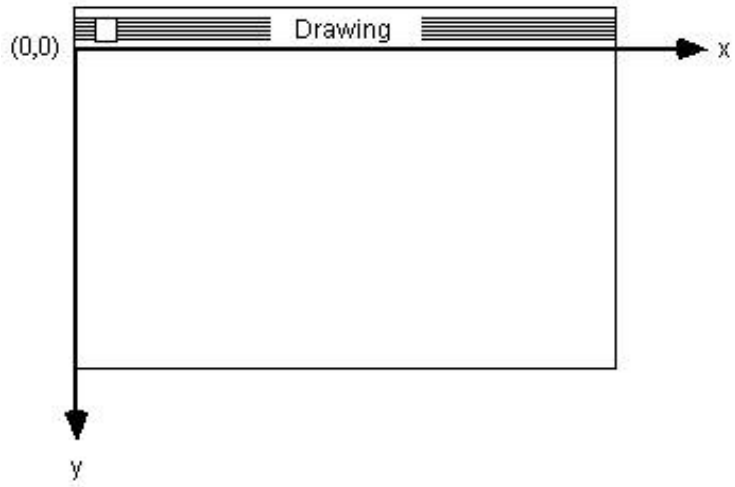
### 2.3.1 **Considérations techniques**

Selon le point 2 de la marche à suivre (cf. 1.7) il faut déterminer les points principaux à traiter. Notre problème étant géométrique, il faut tout d'abord connaître la "surface de dessin" à savoir l'écran du Macintosh.

La documentation technique nous apprend que:

- l'écran du Macintosh est une grille de 512x342 points
- Le dessin s'effectue dans la fenêtre de dessin (Drawing)
- le système de coordonnées (cartésiennes) a son origine au point (0, 0) situé en haut à gauche de la fenêtre de dessin

- les axes sont disposés ainsi:




Ceci établi nous pouvons remarquer qu'il faudra:

- connaître comment faire apparaître la fenêtre de dessin;
- savoir dessiner un carré et un triangle, ou au moins des segments;
- également savoir dessiner (écrire) un texte.

Ces renseignements constituent le point 2 de la marche à suivre.

[ Retour à la table des matières ]

### 2.3.2 **Algorithme de résolution**

L'algorithme de résolution (point 3, cf. 1.7) va être déterminé en utilisant notre méthode. Etant donné le problème initial, on en extrait les sous-problèmes

a. ouvrir (faire apparaître) la fenêtre de dessin
b. dessiner le carré et son titre
c. dessiner le triangle isocèle et son titre

Ceci constitue le premier raffinement. Comme le point *a* est immédiatement traduisible en Pascal Macintosh, laissons-le tel quel. Le raffinement suivant est alors:

a. ouvrir (faire apparaître) la fenêtre de dessin

b.1 dessiner le titre CARRE
b.2 dessiner le carré avec les côtés parallèles aux axes de coordonnées

c.1 dessiner le titre TRIANGLE
c.2 dessiner le triangle isocèle sur la pointe, avec la base parallèle à l'axe des x

Afin de présenter un troisième raffinement, nous supposons que Pascal Macintosh ne permet pas de dessiner directement un carré ou un triangle. On a donc:

a. ouvrir (faire apparaître) la fenêtre de dessin

b.1.1 choisir la position du titre (coin en bas à gauche de la première lettre): (30, 100)
b.1.2 dessiner le mot CARRE depuis cette position

b.2.1 choisir la position du carré (sommet en haut à gauche): (30, 120)

b.2.2 dessiner le côté supérieur depuis cette position
b.2.3 dessiner le côté droit depuis l'extrémité droite du côté supérieur
b.2.4 dessiner le côté inférieur depuis l'extrémité inférieure du côté droit
b.2.5 dessiner le côté gauche depuis l'extrémité gauche du côté inférieur

c.1.1 choisir la position du titre (coin en bas à gauche de la première lettre): (110, 100)
c.1.2 dessiner le mot TRIANGLE depuis cette position

c.2.1 choisir la position du triangle isocèle (sommet gauche): (110, 120)
c.2.2 dessiner la base depuis cette position
c.2.3 dessiner le côté droit depuis l'extrémité droite de la base
c.2.4 dessiner le côté gauche depuis l'extrémité inférieure du côté droit

Cette suite de raffinements s'arrête ici. En effet Pascal Macintosh offre les moyens de

- dessiner un segment (côté) depuis un point courant (une extrémité)
- dessiner un mot (titre)

La suite d'opérations a, b.1.1, b.1.2, ..., c.2.4 est un algorithme de résolution du problème donné, écrit en français (point 3 résolu).

[ Retour à la table des matières ]

### 2.3.3 **Codage de l'algorithme en Pascal**

Le premier essai de codage de l'algorithme en Pascal (point 4, cf. 1.7) nous donne le programme suivant:

```
program exemple_essai_1 (input, output);
begin
   showdrawing;
   moveto (30, 100);
   drawstring ('CARRE');
   moveto (30, 120);
   line (50, 0);
   line (0, 50);
   line (- 50, 0);
   line (0, - 50);
   moveto (110, 100);
   drawstring ('TRIANGLE');
   moveto (110, 120);
   line (80, 0);
   line (- 40, 40);
   line (- 40, - 40);
end.
```

Ce programme est correct, simple, fonctionne parfaitement mais est **horriblement mal écrit**. En effet il faut savoir ou se rappeler que

- un programme doit être **lisible**, ce qu'il n'est pas
- un programme doit être **commenté**, ce qu'il n'est pas
- un programme doit refléter les **décisions prises** par le programmeur (icigrandeur des dessins, points initiaux des dessins)
- les nombres entiers peuvent signifier n'importe quoi! Il faut préciser leur signification en leur substituant des **noms parlants**.

Après avoir tenu compte de ces remarques, un bon programme est:

```
program exemple_bien_fait (input, output);

(* Auteur : Dupont Jean ET1                              *)
(* Date : 10 novembre 1987                               *)
(* Pour la suite, voir le document "Présentation des     *)
(* programmes"                                           *)

    const droite = 1;  (* pour se déplacer d'une unité *)
          gauche = -1; (* dans les quatre directions   *)
          haut   = -1;
          bas    =  1;

          a_l_horizontale = 0; (* Pour un trait vertical, le      *)
                               (* déplacement horizontal est nul  *)
          a_la_verticale = 0;  (* Pour un trait horizontal, le    *)
                               (* déplacement vertical est nul    *)
          demi_base = 40;      (* longueur de la demi-base du     *)
                               (* triangle                        *)
          cote_carre = 50;     (* longueur d'un côté du carré     *)
          distance = 20;       (* distance titre - forme dessinée *)

   var abscisse_titre_carre : integer;    (* abscisse et ordonnée du point    *)
       ordonnee_titre_carre : integer;    (* initial de dessin du titre CARRE *)
       abscisse_titre_triangle : integer; (* abscisse et ordonnée du point    *)
       ordonnee_titre_triangle : integer; (* initial du titre TRIANGLE        *)

begin (* exemple_bien_fait *)
   (* Présentation du programme, en commençant par ouvrir la fenêtre de texte *)
   showtext;
   writeln ( 'Bonjour. Je vais dessiner un carre et un triangle avec leur titre.' );

   (* L'utilisateur donne le point initial de dessin du carré *)
   write ( 'Donnez l''abscisse puis l''ordonnee du point initial du carre:' );
   readln ( abscisse_titre_carre, ordonnee_titre_carre );

   (* Ecriture du titre du dessin du carré, après ouverture de la fenêtre de dessin *)
   showdrawing;
   moveto ( abscisse_titre_carre, ordonnee_titre_carre );
   drawstring ( 'CARRE' );

   (* Dessin du carré *)
   moveto ( abscisse_titre_carre, ordonnee_titre_carre + distance * bas );
   line ( cote_carre * droite, a_l_horizontale );
   line ( a_la_verticale, cote_carre * bas );
   line ( cote_carre * gauche, a_l_horizontale );
   line ( a_la_verticale, cote_carre * haut );

   (* L'utilisateur donne le point initial de dessin du triangle *)
   write ( 'Donnez l''abscisse puis l''ordonnee du point initial du triangle:' );
   readln ( abscisse_titre_triangle, ordonnee_titre_triangle );

   (* Ecriture du titre du dessin du triangle *)
   moveto ( abscisse_titre_triangle, ordonnee_titre_triangle );
   drawstring ( 'TRIANGLE' );
```

```
    (* Dessin du triangle *)
    moveto ( abscisse_titre_triangle, ordonnee_titre_triangle + distance * bas );
    line ( 2 * demi_base * droite, a_l_horizontale );
    line ( demi_base * gauche, demi_base * bas );
    line ( demi_base * gauche, demi_base * haut );
end.
```

[ Retour à la table des matières ]

## 2.4 **Structure d'un programme Pascal**

Un programme Pascal est composé de trois parties principales:

- l'**en-tête** du programme où sont spécifiés le nom du programme ainsi que les fichiers externes utilisés (cf. 10.3.5). L'en-tête est toujours accompagné d'un commentaire précisant le nom de l'auteur, la date de création ... (cf. document "Présentation des programmes").

  Exemple:

```
program exemple_bien_fait (input, output);
(* Auteur  : Dupont Jean ET1              *)
(* Date    : 10 novembre 1986             *)
(* ... ...                                *)
```

- la **partie déclarative** contenant les **déclarations** des objets (constantes, variables, ...) utilisés dans le programme. Ces objets représenteront les données traitées par le programme.

  Notons que les constantes doivent être déclarées avant les variables.

- le **corps** du programme, compris entre les mots **begin** et **end**. Ce corps contient les **instructions** du programme, c'est-à-dire les actions à entreprendre sur les données. C'est principalement dans le corps que se retrouve l'algorithme choisi pour écrire le programme.

[ Retour à la table des matières ]

## 2.5 **Constituants d'un programme**

### 2.5.1 **Généralités**

Un programme est écrit dans un langage (de programmation). Ce langage est composé de mots, symboles, commentaires... Ceux-ci sont groupés en phrases (dont l'ensemble compose le programme) qui obéissent à des règles. Ces règles déterminent de manière absolument stricte si une phrase est correcte ou non (respect de la syntaxe).

L'analogie avec les langues naturelles (français, allemand...) est donc forte, la principale différence étant qu'une phrase française peut être formée de manière beaucoup moins rigoureuse et signifier néanmoins quelque chose. Or une phrase en Pascal doit être absolument juste pour être comprise par le compilateur ou l'interprète, sans quoi elle est obligatoirement rejetée!

Un programme Pascal est composé de phrases appelées **unités syntaxiques.** Elles sont elles-mêmes constituées de mots, symboles... appelés **unités lexicales**. Chaque unité lexicale est une suite de caractères appartenant au jeu de caractères Pascal.

[ Retour à la table des matières ]

### 2.5.2 **Le jeu de caractères Pascal**

Le jeu de caractères Pascal comporte les caractères donnés par l'annexe B. Ajoutons que les constantes caractère et chaînes de caractères (cf. 2.5.3) peuvent être formées de n'importe quel caractère ASCII (cf. annexe A) imprimable.

[ Retour à la table des matières ]

### 2.5.3 **Les unités lexicales de Pascal**

Les unités lexicales en Pascal sont:

- les **identificateurs**, comme par exemple a, *limite_100, cote_carre, write...* Un identificateur est un mot composé de lettres, de chiffres et du caractère "_" et commençant obligatoirement par une lettre. S'il n'y a aucune limite théorique au nombre de caractères composant un identificateur, tous les compilateurs imposent cependant un nombre maximal (par exemple Pascal Macintosh: 255, Pascal VAX : 31).

  Le langage contient des **identificateurs prédéfinis** comme write, writeln, integer... que le programmeur peut éventuellement redéfinir mais dont le but principal reste d'être utilisés **tels quels** dans différentes situations. L'annexe D en donne la liste.

- les **mots réservés** (ou mots-clés) qui sont des identificateurs réservés à un usage bien défini (toujours le même!), par exemple **program const begin .**
  L'annexe C en donne la liste.

- les **symboles** formés d'un unique caractère ou de deux caractères accolés, comme ; = ( := <> par exemple.
  L'annexe B en donne la liste.

- les **commentaires,** c'est-à-dire n'importe quels caractères entre les symboles (* et *) .

- les **constantes numériques** (entières ou réelles) comme 1 123 -36 12.3 124e3 -234.0e3 -0.3e-2 (cf. 3.2 et 3.3).

- les **constantes caractères,** c'est-à-dire un caractère entre apostrophes; par exemple: 'a' '?' 'A' (cf. 4.5).

- les **constantes chaînes de caractères**, c'est-à-dire une suite de plusieurs caractères entre apostrophes; par exemple 'abcd' 'CARRE' 'triangle de Pascal' (cf. 8.2).

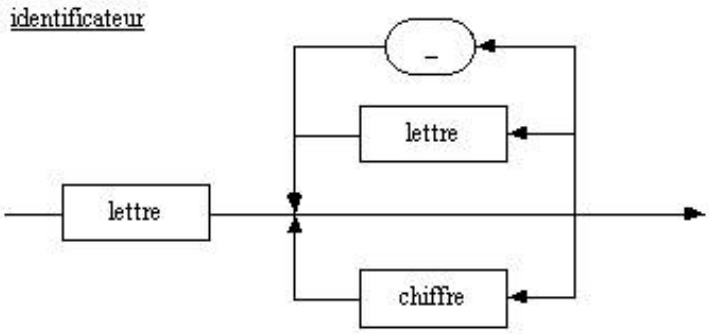
Chaque unité lexicale peut (doit) être **séparée des autres** par un ou plusieurs espaces, caractères de tabulation ou passages à la ligne (examiner les espaces du programme *exemple_bien_fait* cf. 2.3.3).

[ Retour à la table des matières ]

### 2.5.4 L**es unités syntaxiques de Pascal et diagrammes syntaxiques**

Les unités syntaxiques en Pascal peuvent être décrites par des diagrammes appelés **diagrammes syntaxiques**. L'annexe G en donne la liste.

Voici l'exemple d'un tel diagramme permettant de vérifier qu'un "mot" est bien un identificateur:




[ Retour à la table des matières ]

## 2.6 **Mise en page**

La mise en page d'un programme Pascal est assez libre. Cependant, pour des raisons de clarté, de lisibilité d'un programme il existe des conventions qu'il faut respecter. Elles sont décrites dans le document "Présentation des programmes".

[ Retour à la table des matières ]

**CHAPITRE 10**

# LES FICHIERS

## 10.1 **Motivation**

Toutes les structures de données que nous avons vues ont un point commun: elles résident toutes dans la mémoire principale de l'ordinateur. Ceci signifie que l'effacement (volontaire ou non!) de la mémoire provoque la destruction de ces structures, ainsi d'ailleurs que celle du programme Pascal les utilisant.

D'autre part il peut être nécessaire de conserver certaines données après la fin du programme les ayant créées, ceci en prévision d'une utilisation future.

Ces considérations nous amènent à introduire la notion de **fichier**.

[ Retour à la table des matières ]

## 10.2 **Qu'est-ce qu'un fichier?**

Hors du monde informatique un fichier est une collection de fiches; chacun de nous a déjà manipulé un fichier dans une bibliothèque, une administration etc. Ces fichiers se caractérisent par un nombre quelconque de fiches en général toutes de même aspect. Pour trouver une fiche il faut parcourir le fichier fiche après fiche ou utiliser une clé d'accès si le fichier est trié selon cette clé (ordre alphabétique, ordre de cotation des livres...).

En informatique on définit qu'un **fichier** est une structure de données toutes de même type mais dont le nombre n'est pas connu à priori. L'accès à un élément (à une donnée) du fichier se fait

- séquentiellement c'est-à-dire en parcourant le fichier élément par élément depuis le début jusqu'à l'élément choisi

- directement en donnant la position de l'élément cherché

- selon une clé chaque valeur de la clé désignant un élément on obtient ainsi l'élément désiré

Les fichiers sont conservés en **mémoire secondaire** (disques et bandes magnétiques, disquettes, cassettes...) et subsistent tant que cette mémoire secondaire n'est pas effacée ou endommagée. Chaque fichier est désigné par un nom et possède des attributs tels que date de création, taille, icône (Macintosh)...

Ils se répartissent en deux catégories:

1. Les **fichiers binaires** contenant du code binaire représentant chaque élément (cf. 10.3). Ces fichiers ne doivent être manipulés que par des programmes!

2. Les **fichiers de texte** (appelés aussi imprimables) contenant des caractères et susceptibles d'être lus, éditées, imprimés... (cf. 10.4).

Pascal va permettre de créer et d'utiliser des fichiers à accès séquentiel uniquement. Notons cependant que de nombreuses versions de Pascal offrent des possibilités d'accès direct, voire d'accès par clé.

**En particulier** : Pascal Macintosh permet l'accès direct, Pascal VAX les accès direct et par clé.

[ Retour à la table des matières ]

## 10.3 **Les types fichier**

### 10.3.1 **Généralités**

Un fichier (binaire) se déclare au moyen des mots réservés **file of**. Les types fichier sont structurés, le type de leurs éléments peut être n'importe lequel (sauf un type fichier!). Comme d'habitude une variable fichier sera définie au moyen d'un type fichier.

**Exemples:**

```
    const long_max_nom = 30;        (* pour le type t_auteur_livre *)
          long_max_titre = 40;

    type  t_fich_entiers = file of integer;
          t_fich_reels = file of real;

          t_auteur_livre = string [ long_max_nom ];  (* syntaxe Pascal Macintosh! *)
          t_titre_livre = string [ long_max_titre ];

          t_livre_cote = record          (* représente une fiche bibliographique *)
                           nom_auteur : t_auteur_livre;
                           titre : t_titre_livre;
                           cote : integer;
                         end;

          t_fichier_biblio = file of t_livre_cote;

     var  fichier_reponses : t_fich_entiers;    (* fichier dont les éléments seront
*)
                                                 (* tous entiers *)

          fichier_mesures : t_fich_reels;        (* contient les mesures du 16.2.87
*)
                                                 (* de la température diurne (en Co)
*)

          bibliotheque : t_fichier_biblio;       (* ouvrages de la bibliothèque de *)
                                                 (* l'EINEV jusqu'au 31.12.1990 *)
```

Il existe un type fichier prédéfini: *text* (cf. 10.4).

Il existe deux (variables) fichiers prédéfinis: *input* et *output* (cf. 10.4) de type *text* .

La seule **opération** permise sur les fichiers est le passage en paramètre: un fichier doit toujours être un paramètre d'entrée et de sortie! L'affectation est donc interdite entre fichiers. Notons qu'en général les fichiers sont mentionnés comme paramètres de tout programme les utilisant (cf. 10.3.5).

Les **expressions** sont réduites aux variables d'un type fichier.

Comme dans le cas des tableaux (cf. 6.3) l'intérêt des fichiers réside en l'utilisation de leurs éléments (cf. 10.3.3).

[ Retour à la table des matières ]

### 10.3.2 **Ouverture et fermeture des fichiers**

En Pascal un fichier peut être soit lu soit écrit. On dit alors que le fichier est **ouvert en lecture respectivement en écriture**. Pour utiliser un fichier celui-ci doit donc préalablement être ouvert soit en lecture soit en écriture. Ceci se fait

- par la procédure prédéfinie *reset* (cf. 10.3.4) pour l'ouverture en lecture

- par la procédure prédéfinie *rewrite* (cf. 10.3.4) pour l'ouverture en écriture.
  Notons que l'ouverture en écriture provoque l'initialisation du fichier!

**Exemple:**

```
var bibliotheque : t_fichier_biblio; (* cf. 10.3.1 *)

begin
     rewrite ( bibliotheque );          (* initialisation du fichier bibliotheque *)
     ...                                (* écriture dans le fichier *)
     reset ( bibliotheque );            (* le fichier bibliotheque va être relu *)
     ...                                (* lecture du fichier *)
end.
```

La fermeture d'un fichier est faite automatiquement à la fin du programme. Attention cependant à certaines versions de Pascal qui exigent une fermeture explicite, rendant ces fichiers inutilisables par la suite en cas d'oubli! Pascal Macintosh et Turbo Pascal exigent une fermeture avant une réouverture du même fichier dans un autre mode. Cette fermeture s'effectue par la procédure *close* (cf. 10.3.4).

[ Retour à la table des matières ]

### 10.3.3 **Accès séquentiel aux éléments d'un fichier**

L'accès à un élément va consister à le lire ou à l'écrire. Pour faciliter la compréhension il faut s'imaginer que

- une fenêtre est ouverte sur cet élément et en permet l'accès alors que tous les autres sont cachés.

- tout fichier est terminé par une marque de fin de fichier permettant de savoir où se termine le fichier.

L'accès séquentiel est le fait que:

- La lecture d'un élément se fait en parcourant le fichier depuis le début jusqu'à l'élément. La fenêtre désigne alors l'élément à lire.

- L'écriture d'un élément se fait toujours à la fin du fichier. La fenêtre désigne alors l'endroit où va être écrit l'élément.

**Exemple:**

```
var bibliotheque : t_fichier_biblio; (* cf. 10.3.1 *)
```

alors la fenêtre se note

```
bibliotheque^
```

Elle s'utilise comme une variable de type *t_livre_cote* (cf. 10.3.1). Il est donc autorisé d'accéder aux différents champs ainsi:

```
bibliotheque^.nom_auteur                (* de type t_auteur_livre *)
bibliotheque^.titre                     (* de type t_titre_livre *)
bibliotheque^.cote                      (* de type integer *)
```

Pour un fichier ouvert en lecture la fenêtre est avancée d'un élément au moyen de la procédure prédéfinie *get* (cf. 10.3.4) alors que pour un fichier ouvert en écriture l'élément est véritablement écrit et la fenêtre avancée par la procédure prédéfinie *put* (cf. 10.3.4).

**Exemples:**

Soit la variable livre de type *t_livre_cote* (cf. 10.3.1). Alors

- si le fichier *bibliotheque* est ouvert en lecture:

```
livre := bibliotheque^;  (* copier la valeur de l'élément courant dans livre *)
get ( bibliotheque );    (* positionner la fenêtre sur l'élément suivant *)
```

- si le fichier *bibliotheque* est ouvert en écriture:

```
bibliotheque^ := livre;  (* copier la valeur de livre dans la fenêtre *)
put ( bibliotheque );    (* écrire la valeur contenue dans la fenêtre et *)
                         (* avancer celle-ci *)
```

[ Retour à la table des matières ]

### 10.3.4 **Procédures et fonctions prédéfinies**

Soit la variable *exemple* d'un type fichier quelconque. Les procédures applicables à exemple sont les suivantes:

```
  rewrite ( exemple );  ouvre le fichier exemple en écriture et l'initialise à la
valeur
                        "fichier vide" (le fichier ne contient que la marque de fin
de
                        fichier).La fonction eof ( exemple ) est alors vraie.

  reset ( exemple );    ouvre le fichier exemple en lecture et positionne la fenêtre
                        exemple^ sur le premier élément. Si le fichier est vide la
                        fonction eof ( exemple ) est vraie.

  put ( exemple );      écrit l'élément contenu dans exemple^ à la fin du fichier. Il
                        faut s'imaginer que la marque de fin est alors placée après
cet
                        élément et que exemple^ est positionnée sur cette marque.

  get ( exemple );      positionne exemple^ sur l'élément suivant du fichier. La
                        fonction eof ( exemple ) devient vraie si cet élément est la
                        marque de fin de fichier.
```

La fonction applicable à *exemple* est la suivante:

```
  eof ( exemple )       donne la valeur true si la fenêtre exemple^  désigne la
marque
                        de fin de fichier, false sinon.
```

De nombreux dialectes offrent également la procédure

```
  close ( exemple );    ferme le fichier exemple .
```

[ Retour à la table des matières ]

### 10.3.5 **Fichiers internes et externes**

Pascal distingue deux catégories de fichiers

- les fichiers externes (ou permanents) qui subsistent après la fin de tout programme les utilisant. Un fichier est rendu externe s'il est mentionné comme paramètre du programme au même titre que *input* et *output*.

  **Exemple:**

```
program copie_de_fichiers (input, output, original, copie);
...
   type   t_fichier = file of ...;
   var    original : t_fichier;   (* deux fichiers externes *)
          copie : t_fichier;
...
```

- les fichiers internes (ou temporaires) qui sont détruits automatiquement à la fin du programme. Un fichier est rendu interne s'il **n'est pas** mentionné comme paramètre du programme.

  **Exemple:**

```
program fichiers_internes (input, output);
...
  type   t_fichier = file of ...;
  var    resultat : t_fichier;    (* deux fichiers internes *)
         copie : t_fichier;
 ...
```

**Remarque importante :**

Pascal Macintosh traite différemment les fichiers internes et externes:

- aucun fichier (autre que *input* et *output* ) ne doit figurer comme paramètre d'un programme.

- un fichier est externe si et seulement s'il est associé à un fichier sur disquette, disque dur... Cette association se fait lors de l'appel à *reset* ou *rewrite*.

**Exemples:**

reset ( original, 'Exercice 3');
rewrite ( copie, 'Exercice 3 bis');

[ Retour à la table des matières ]

### 10.3.6 **Exemple complet**

Voici un programme permettant de copier un fichier dans un autre.

```
program copie_de_fichiers (input, output, original, copie);
(* Auteur: ... *)

        const   long_max_nom = 30; (* nombre maximal de caractères du nom d'un auteur
*)

        type  t_auteur_livre = string [ long_max_nom ]; (* syntaxe Pascal Macintosh!
*)
```

```
                t_livre_cote = record          (* représente une fiche bibliographique
*)
                                          nom_auteur : t_auteur_livre; (* nom de
l'auteur *)
                                          cote : integer;              (* cote en
bibliothèque *)
                                          annee : integer;             (* année de
parution *)
                                end;

                t_fichier_biblio = file of t_livre_cote;

        var     original : t_fichier_biblio;   (* fichier à copier *)
                 copie : t_fichier_biblio;      (* copie à créer *)

        begin (* copie_de_fichiers *)

           reset ( original );     (* ouvrir le fichier à copier *)
           rewrite ( copie );      (* initialiser la copie *)

           while not eof ( original ) do   (* parcourir les éléments du fichier à
copier *)
           begin
               copie^ := original^;          (* copier l'élément courant d'une
fenêtre *)
                                             (* dans l'autre *)
               put ( copie );                (* écrire l'élément copié *)
               get ( original );             (* passer à l'élément à copier suivant
*)
           end;
       (* fermeture automatique en fin de programme *)
        end.
```

**Remarque :**

Les fichiers *input* et *output* ne sont pas utilisé dans ce programme. Il serait alors possible de les omettre dans l'en-tête du programme. Les fichiers (externes) utilisés ont les noms *original* et *copie* .

[ Retour à la table des matières ]

## 10.4 **Les fichiers de texte**

### 10.4.1 **Généralités**

Les fichiers de texte sont des cas particuliers de fichiers. Un fichier de texte est formé d'éléments bien connus: les caractères. Chacun a déjà manipulé de tels fichiers: un programme (Pascal ou autre) est en fait un fichier de texte!

Les caractères contenus dans un fichier de texte sont organisés en lignes, chacune terminée par une **marque de fin de ligne**. Après la dernière ligne, le fichier se termine par une **marque de fin de fichier**.

Tout ce qui a été dit est valable pour les fichiers de texte. Précisons simplement que

1. Un fichier est un fichier de texte s'il est déclaré au moyen du type prédéfini *text* .

2. Les procédures prédéfinies read, readln, write, writeln sont utilisables avec n'importe quel fichier de texte. Il suffit alors de

rajouter le nom de ce fichier comme premier paramètre.

**Exemples :** read (nom_de_fichier_1,v2,...vn)
write (nom_de_fichier_2, e1, e2, ... en );

Leur comportement est strictement identique à celui observé lors d'une lecture au clavier ou d'une écriture à l'écran!

3. Soient *ch* une variable caractère et *f* un fichier de texte. Les instructions

| | | | |
|---|---|---|---|
| read ( f, ch ); | resp. | write ( f, ch ); de | sont en fait les abréviations |
| ch := f`; | resp. | f^ := ch;<br>put ( f ); | |

4. Il existe deux fichier de texte prédéfinis: *input* et *output* . En fait ces deux fichiers représentent le clavier (*input*, considéré comme ouvert en lecture) et l'écran ( *output* , considéré comme ouvert en écriture). On a jusqu'à présent utilisé (sans le savoir) les abréviations suivantes:

| | | |
|---|---|---|
| read ( v1, v2 ..., vn ); | pour | read ( input, v1, v2, ..., vn ); |
| write ( e1, e2, ..., en ); | pour | write ( output, e1, e2, ..., en ); |
| eof | pour | eof ( input ) |
| eoln | pour | eoln ( input ) |

5. Il existe la procédure prédéfinie

   page ( nom_fichier_texte );

   qui place une marque de page dans le fichier *nom_fichier_texte* . Cette marque provoquera un saut de page à cet endroit lors de l'impression de ce fichier. Il est déconseillé de l'utiliser avec le fichier prédéfini *output* !

**Remarque:**

Un fichier de type text n'est généralement pas équivalent à un fichier de "type" **file of** *char* qui, lui, ne possède pas une structure de lignes!

[ Retour à la table des matières ]

### 10.4.2 **Traitement d'un fichier de texte**

Le traitement d'un fichier de texte au sens le plus général consiste en la lecture, le traitement proprement dit et l'écriture d'un (nouveau) fichier de texte.

Une lecture correcte doit tenir compte des marques de fin de ligne et de fichier sous peine de provoquer des erreurs. Pour cela le programmeur utilise deux fonctions booléennes prédéfinies:

- eoln ( nom_fichier_texte ) vraie si le **prochain** caractère à lire (dans le fichier *nom_fichier_texte* ) est la marque de fin de ligne symbolisée dans les exemples qui suivent par Ω

- eof ( nom_fichier_texte ) vraie si le **prochain** caractère à lire (dans le fichier *nom_fichier_texte* ) est la marque de fin de fichier symbolisée dans les exemples qui suivent par •

**Exemple de fichier de texte:**

ABC 12Ω
3c de Ω
Ω
23tropΩ
•

Un algorithme juste effectuant la lecture et le traitement est par exemple:

**tant que** il y a encore des lignes avant • **faire**
**debut**

  **tant que** il y a encore des caractères avant Ω **faire**
  **debut**
    lire le prochain caractère
    traiter le caractère lu
  **fin**

  sauter la marque de fin de ligne Ω
**fin**

Cet algorithme traduit en Pascal donne (on suppose lire le fichier de texte *exemple* ):

| version lecture caractère par caractère | version lecture ligne par ligne |
|---|---|
| **while not** eof ( exemple ) **do**<br>**begin**<br>  **while not** eoln ( exemple ) **do**<br>  **begin**<br>  read ( exemple, caractere );<br>  ... (* traiter le caractère lu *)<br>  **end;**<br>  readln ( exemple );<br>**end;** | **while not** eof ( exemple ) **do**<br>**begin**<br>  readln ( exemple, chaine_caracteres );<br>  ... (* traiter le chaîne lue *)<br>**end;** |

**Remarques:**

1. Au clavier, la marque de fin de ligne Ω peut être introduite en tapant la touche <ret>, la marque de fin de fichier • en tapant les touches

   - <enter> sur Macintosh
   - <ctrl>z sur VAX/VMS

2. L'instruction read ( *exemple, caractere* ); a l'effet suivant **si le "caractère" à lire est la marque de fin de ligne Ω :**

```
caractere := ' ';        (* la variable caractere contient un espace *)
readln ( exemple );      (* positionnement au début de la ligne suivante *)
```

Comme le résultat de cette lecture varie selon les dialectes de Pascal, il ne faudrait jamais lire la marque de fin de ligne mais toujours utiliser la fonction eof!

[ Retour à la table des matières ]

### 10.4.3 **Comportement des fonctions *eof* et *eoln***

Le langage Pascal est **normalisé** depuis 1983. Cependant de nombreuses versions de Pascal sont disponibles, répandues sur de nombreux ordinateurs différents. Ceci signifie que toutes ces versions de Pascal diffèrent sur quelques points, en particulier sur la **sémantique des fonctions prédéfinies *eof* et *eoln* .** Nous allons la préciser ici pour les versions de Pascal Macintosh et Pascal VAX.

Cette sémantique est identique pour ces deux versions, à savoir:

```
var entier : integer;     (* pour notre exemple *)
    reel  : real;
    caractere : char;
    exemple   : text;     (* supposé ouvert en lecture *)
```

| | |
|---|---|
| 1. Après l'instruction *read ( exemple, entier );* | la fonction *eoln* est **vraie** si la marque de fin de ligne suit le nombre entier lu. |
| 2. Après l'instruction *read (exemple, reel );* | même comportement de *eoln* que sous 1. |
| 3. Après l'instruction *read ( exemple, caractere );* | la fonction *eoln* est **vraie** si le prochain "caractère" est la marque de fin de ligne. |
| 4. Après l'instruction *readln ( exemple );* | la fonction *eof* est **vraie** si la marque de fin de fichier suit la marque de fin de ligne. La fonction *eoln* est **vraie** si la ligne est vide. |

**Remarque:**

La fonction *eoln* est immédiatement vraie si une ligne est vide.

[ Retour à la table des matières ]

**ANNEXE A**

# CODE ASCII

| Caractère | Code | | | Caractère | Code | | | Caractère | Code | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Déc. | Oct. | Hex. | | Déc. | Oct. | Hex. | | Déc. | Oct. | Hex. |
| *NUL* | 0 | 0 | 0 | + | 43 | 53 | 2B | | | | |
| *SOH* | 1 | 1 | 1 | , | 44 | 54 | 2C | V | 86 | 126 | 56 |
| *STX* | 2 | 2 | 2 | - | 45 | 55 | 2D | W | 87 | 127 | 57 |
| *ETX* | 3 | 3 | 3 | . | 46 | 56 | 2E | X | 88 | 130 | 58 |
| *EOT* | 4 | 4 | 4 | / | 47 | 57 | 2F | Y | 89 | 131 | 59 |
| *ENQ* | 5 | 5 | 5 | 0 | 48 | 60 | 30 | Z | 90 | 132 | 5A |
| *ACK* | 6 | 6 | 6 | 1 | 49 | 61 | 31 | [ | 91 | 133 | 5B |
| *BEL* | 7 | 7 | 7 | 2 | 50 | 62 | 32 | \ | 92 | 134 | 5C |
| *BS* | 8 | 10 | 8 | 3 | 51 | 63 | 33 | ] | 93 | 135 | 5D |
| *HT* | 9 | 11 | 9 | 4 | 52 | 64 | 34 | ^ | 94 | 136 | 5E |
| *LF* | 10 | 12 | A | 5 | 53 | 65 | 35 | _ | 95 | 137 | 5F |
| *VT* | 11 | 13 | B | 6 | 54 | 66 | 36 | ` | 96 | 140 | 60 |
| *FF* | 12 | 14 | C | 7 | 55 | 67 | 37 | a | 97 | 141 | 61 |
| *CR* | 13 | 15 | D | 8 | 56 | 70 | 38 | b | 98 | 142 | 62 |
| *SO* | 14 | 16 | E | 9 | 57 | 71 | 39 | c | 99 | 143 | 63 |
| *SI* | 15 | 17 | F | : | 58 | 72 | 3A | d | 100 | 144 | 64 |
| *DLE* | 16 | 20 | 10 | ; | 59 | 73 | 3B | e | 101 | 145 | 65 |
| *DC1* | 17 | 21 | 11 | < | 60 | 74 | 3C | f | 102 | 146 | 66 |
| *DC2* | 18 | 22 | 12 | = | 61 | 75 | 3D | g | 103 | 147 | 67 |
| *DC3* | 19 | 23 | 13 | > | 62 | 76 | 3E | h | 104 | 150 | 68 |
| *DC4* | 20 | 24 | 14 | ? | 63 | 77 | 3F | i | 105 | 151 | 69 |
| *NAK* | 21 | 25 | 15 | @ | 64 | 100 | 40 | j | 106 | 152 | 6A |
| *SYN* | 22 | 26 | 16 | A | 65 | 101 | 41 | k | 107 | 153 | 6B |
| *ETB* | 23 | 27 | 17 | B | 66 | 102 | 42 | l | 108 | 154 | 6C |
| *CAN* | 24 | 30 | 18 | C | 67 | 103 | 43 | m | 109 | 155 | 6D |
| *EM* | 25 | 31 | 19 | D | 68 | 104 | 44 | n | 110 | 156 | 6E |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| *SLB* | 26 | 32 | 1A | E | 69 | 105 | 45 | o | 111 | 157 | 6F |
| *ESC* | 27 | 33 | 1B | F | 70 | 106 | 46 | p | 112 | 160 | 70 |
| *FS* | 28 | 34 | 1C | G | 71 | 107 | 47 | q | 113 | 161 | 71 |
| *GS* | 29 | 35 | 1D | H | 72 | 110 | 48 | r | 114 | 162 | 72 |
| *RS* | 30 | 36 | 1E | I | 73 | 111 | 49 | s | 115 | 163 | 73 |
| *US* | 31 | 37 | 1F | J | 74 | 112 | 4A | t | 116 | 164 | 74 |
| space | 32 | 40 | 20 | K | 75 | 113 | 4B | u | 117 | 165 | 75 |
| ! | 33 | 41 | 21 | L | 76 | 114 | 4C | v | 118 | 166 | 76 |
| " | 34 | 42 | 22 | M | 77 | 115 | 4D | w | 119 | 167 | 77 |
| # | 35 | 43 | 23 | N | 78 | 116 | 4E | x | 120 | 170 | 78 |
| $ | 36 | 44 | 24 | O | 79 | 117 | 4F | y | 121 | 171 | 79 |
| % | 37 | 45 | 25 | P | 80 | 120 | 50 | z | 122 | 172 | 7A |
| & | 38 | 46 | 26 | Q | 81 | 121 | 51 | { | 123 | 173 | 7B |
| ' | 39 | 47 | 27 | R | 82 | 122 | 52 | \| | 124 | 174 | 7C |
| ( | 40 | 50 | 28 | S | 83 | 123 | 53 | } | 125 | 175 | 7D |
| ) | 41 | 51 | 29 | T | 84 | 124 | 54 | ~ | 126 | 176 | 7E |
| * | 42 | 52 | 2A | U | 85 | 125 | 55 | *DEL* | 127 | 177 | 7F |

[ Retour à la table des matières ]

**ANNEXE D**

# IDENTIFICATEURS PREDEFINIS DE PASCAL

| | | |
|---|---|---|
| *abs* | *ln* | *sin* |
| *arctan* | | *sqr* |
| | *maxint* | *sqrt* |
| *boolean* | | *succ* |
| | *new* | |
| *char* | | *text* |
| *chr* | *odd* | *true* |
| *cos* | *ord* | *trunc* |
| | *output* | |
| *dispose* | | *unpack* |
| | *pack* | |
| *eof* | *page* | *write* |
| *eoln* | *pred* | *writeln* |
| *exp* | *put* | |
| | | |
| *false* | *read* | |
| | *readln* | |
| *get* | *real* | |
| | *reset* | |
| *input* | *rewrite* | |
| *integer* | *round* | |

**Remarque:**

Chaque version de Pascal possède des identificateurs prédéfinis supplémentaires!

[ Retour à la table des matières ]

**ANNEXE C**

# MOTS RESERVES DE PASCAL

| | | |
|---|---|---|
| **and** | **goto** | **record** |
| **array** | | **repeat** |
| | **if** | |
| **begin** | **in** | **set** |
| | | |
| **case** | **label** | **then** |
| **const** | | **to** |
| | **mod** | **type** |
| **div** | | |
| **do** | **nil** | **until** |
| **downto** | **not** | |
| | | **var** |
| **else** | **of** | |
| **end** | **or** | **while** |
| **external** | | **with** |
| | **packed** | |
| **file** | **procedure** | |
| **for** | **program** | |
| **forward** | | |
| **function** | | |

**Remarques:**

1. Les mots réservés sont soumis à des conventions d'écriture:

   - sur Macintosh ils apparaissent en **caractères gras**

   - sur VAX il faut les taper en MAJUSCULES, le reste étant écrit en minuscules

- sur papier il faut les écrire en **minuscules soulignées**

2. Pascal Macintosh possède en plus les mots réservés suivants:

**otherwise string uses**

3. Pascal VAX possède en plus les mots réservés suivants:

**module otherwise rem value varying**

**%descr %dictionary %include %immed %ref %stdescr**

[ Retour à la table des matières ]

**ANNEXE B**

# JEU DE CARACTERES ET SYMBOLES DE PASCAL

## JEU DE CARACTERES

Les caractères suivants sont utilisés pour écrire tout programme Pascal:

a) les **chiffres** 0 1 2 3 4 5 6 7 8 9

b) les **lettres** a b c... z A B C... Z

c) les **caractères spéciaux** suivants:

| | |
|---|---|
| ( | parenthèse gauche |
| ) | parenthèse droite |
| [ | crochet gauche |
| ] | crochet droit |
| { | accolade gauche |
| } | accolade droite |
| + | plus |
| - | moins |
| * | fois, étoile |
| / | divisé, slash |
| < | inférieur à (plus petit que) |
| = | égale à |
| > | supérieur à (plus grand que) |
| : | deux-points |
| ; | point-virgule |
| . | point |
| , | virgule |
| ' | apostrophe |
| ^ | chapeau, flèche |

d) le caractère espace appelé aussi blanc
le caractère de tabulation *( HT )*

e) le caractère _ (souligné)

**Remarque:**

Tout caractère imprimable peut appartenir à une constante caractère, à une chaîne de caractères ou à un commentaire!

## SYMBOLES

a) les **symboles à un caractère** sont les caractères spéciaux

b) les **symboles à deux caractères** (accolés) sont les suivants:

| | |
|---|---|
| := | affection |
| .. | intervalle (de...à...) |
| <= | inférieur ou égal à (plus petit ou égal à) |
| <> | différent de |
| >= | supérieur ou égal à (plus grand ou égal à) |
| (* | début commentaire |
| *) | fin de commentaire |

[ Retour à la table des matières ]

**CHAPITRE 3**

# LES NOMBRES ET LES ENTREES-SORTIES DE BASE

---

## 3.1 **Rappels**

Un programme se compose de trois parties:

- en-tête
- partie déclarative
- corps (entre les mots réservés **begin** et **end**)

La partie déclarative contient les déclarations des objets (constantes, variables...) représentant les données (valeurs) que l'on veut traiter.

Le corps contient les instructions du programme, c'est-à-dire les traitements à effectuer sur ces données.

[ Retour à la table des matières ]

## 3.2 **Le type *integer***

### 3.2.1 **Motivation**

Le type *integer* va permettre de traiter les nombres entiers. Ceux-ci sont traditionnellement utilisés les premiers dans un cours d'apprentissage de la programmation. En effet ils ne posent pas (ou peu) de problèmes. Le travail avec eux est naturel puisqu'il correspond à l'arithmétique.
Deux réflexes doivent cependant être acquis le plus vite possible:

- la division entre 2 entiers est une division entière (cf. 3.2.2)
- tous les entiers ne sont évidemment pas utilisables sur un ordinateur; en effet un entier occupe un mot en mémoire et il existe une infinité de nombres entiers (cf. 3.5.2).

[ Retour à la table des matières ]

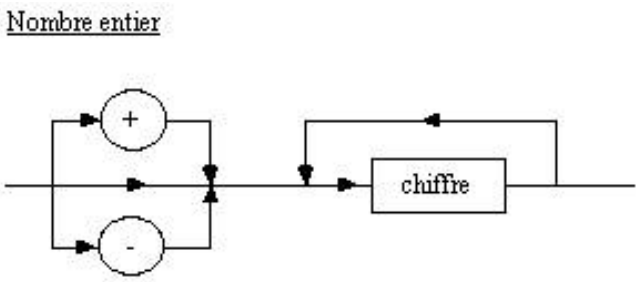
### 3.2.2 **Généralités**

*Integer* est un **identificateur prédéfini**. Il signifie "un objet (constante, variable...) de type *integer* contiendra un nombre entier et rien d'autre". Notons que le type *integer* fait partie des types standard (cf. annexe F).

**Exemples:**

- les constantes droite, gauche... et les variables abscisse_titre_carre... du programme exemple_bien_fait (cf. 2.3.3) sont de type *integer*.

Les nombres entiers doivent respecter la syntaxe suivante:




Les **constantes** de type *integer* sont les nombres entiers eux-mêmes. De plus il existe une constante entière prédéfinie *maxint* (cf. 3.5.2).

Les **opérations** possibles sur les valeurs entières sont:

+ - * **div mod**

où + représente l'addition
- représente la soustraction
* représente la multiplication
**div** est un mot réservé représentant la division entière
**mod** est un mot réservé représentant le reste de la division entière

Les symboles + - * **div mod** sont appelés **opérateurs** (entiers) alors que les valeurs sur lesquelles ils opèrent sont les **opérandes.**

Les **expressions** de type *integer* sont des combinaisons de ces opérations.

**Exemples:**

```
- 1            est une expression réduite à une seule constante
- 3 + 4        est une expression de valeur 7, l'opérateur
               est + et les opérandes sont les constantes 3 et 4
- 4 * 5        est une expression de valeur 20
- 4 div 2      est une expression de valeur 2
- 5 div 2      est une expression de valeur 2
- 4 mod 2      est une expression de valeur 0
- 5 mod 2      est une expression de valeur 1
- 2 + 3 * 4    est une expression qui vaut 14 (cf. 3.4)
- 2 + (3 * 4)  est une expression qui vaut 14
- (2 + 3) * 4  est une expression qui vaut 20
```

Il est possible d'écrire des valeurs entières sur l'écran au moyen de write(...) (cf. 3.6.5).

**Exemples:**

```
- write ( 1 );     écrit la valeur 1 à l'écran
- write ( 4, 10 ); écrit la valeur 4 suivie de la valeur 10 sur l'écran
- write ( 3 + 4 ); écrit la valeur de l'expression, à savoir 7, sur l'écran

- const  max = 10;

  var    nombre : integer;  (* pour notre exemple *)
  ...
     write ( max );          écrit la valeur de max (i. e. 10) sur l'écran
```

```
write ( 2 * max + 1 );  écrit la valeur 21 sur l'écran
write ( max * nombre ); écrit la valeur de max * nombre sur l'écran
```

Lors d'une écriture il est recommandé de toujours faire précéder une valeur par un message (cf. 3.6.4) se rapportant à cette valeur. Par exemple:

write ( 'Le resultat du calcul vaut: ', nombre );

Le message Le resultat du calcul vaut: est écrit suivi de la valeur de nombre (cf. 3.6.5).

[ Retour à la table des matières ]

### 3.2.3 **Affectation**

Le mot affectation signifie "donner une valeur à". **Parler d'affectation de 3 à la variable nombre** signifie "donner la valeur 3 à nombre". Cela s'écrit

```
nombre := 3;       (* pour nombre cf. 3.2.2 *)
```

**Exemples:**

```
- nombre := 5 + 4;            affecte la valeur 9 à nombre
- nombre := (36 div 10) * 2;  affecte la valeur 6 à nombre
- nombre := max;              affecte la valeur de max
                              (cf. 3.2.2) à nombre
```

De manière générale on a:

**nom_de_variable_entière := expression_de_type_integer;**

**Remarque importante:**

Considérons le morceau de programme suivant:

```
var  nombre : integer;   (* pour notre exemple *)
     valeur : integer;

begin
     valeur := nombre + 1;
     ...
```

Que vaut valeur après l'affectation? La réponse est que valeur a une **valeur indéfinie** car la valeur de *nombre* n'était pas définie au moment de l'affectation. La variable *nombre* non initialisée possédait la valeur entière calculée à partir de la suite de bits du mot mémoire correspondant.

L'utilisation de variables déclarées mais non initialisées, c'est-à-dire ne possédant pas de valeur définie au moment de leur utilisation, est une erreur très courante. Ajoutons à cela que la détection de ces erreurs est difficile puisqu'il est possible que le programme s'exécute tout de même, en produisant évidemment n'importe quels résultats.

**Règle**: Le programmeur qui utilise la valeur d'une variable doit être certain que cette valeur est bien définie.

[ Retour à la table des matières ]

### 3.2.4 **Fonctions prédéfinies**

Tout langage de programmation met à disposition du programmeur des **fonctions prédéfinies**. Celles-ci exécutent des actions telles que

- calcul d'un carré
- calcul d'un sinus
- ...

Certaines de ces fonctions peuvent être utilisées dans les expressions arithmétiques entières. L'annexe E en donne la liste complète ainsi que les modalités d'utilisation. Mentionnons simplement

- `abs (valeur)` valeur absolue de *valeur*, par exemple *abs* (-3) vaut 3
- `sqr (valeur)` carré de *valeur*, par exemple *sqr* (5) vaut 25
- `trunc (valeur)` partie entière de *valeur*, par exemple *trunc* (3.14) vaut 3
- `round (valeur)` arrondi de *valeur*, par exemple *round* (1.6) vaut 2

**Remarque**

Le calcul d'une fonction est l'opération la plus prioritaire dans une expression.

[ Retour à la table des matières ]

## 3.3 **Le type real**

### 3.3.1 **Motivation**

Les valeurs de type *real* appelées nombres réels sont indispensables dans les applications dites numériques de la programmation (programmes d'analyse numérique, de statistique, de calcul par éléments finis...).

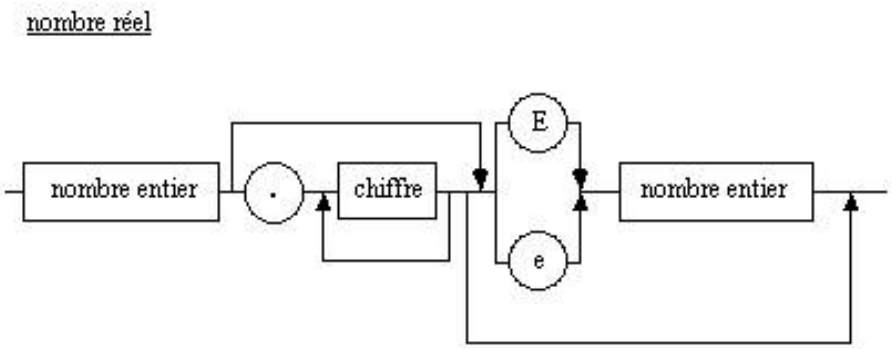
Dans la majorité des autres cas les nombres réels sont peu utilisés, en particulier dans un cours de programmation de base. Leur utilisation sans un minimum de précautions est d'ailleurs dangereuse du fait des erreurs de calcul provoquées par leur représentation en mémoire (cf. 3.5.2).

Nous allons donc examiner ces nombres réels sans trop de détails.

[ Retour à la table des matières ]

### 3.3.2 **Généralités**

*Real* est un **identificateur prédéfini.** Il signifie "un objet (constante, variable...) de type real contiendra un nombre réel et rien d'autre". Un tel nombre doit respecter la syntaxe suivante:




Les **constantes** de type *real* sont les nombres réels eux-mêmes.

Les **opérations** possibles sur les valeurs réelles sont:

```
          +     -      *       /

   où  +   représente l'addition
       -   représente la soustraction
       *   représente la multiplication
       /   représente la division

   Les symboles + - * / sont les opérateurs réels.
```

Les **expressions** de type *real* sont des combinaisons de ces opérations.

**Exemples:**

```
- 1.0              est une expression réduite à une seule constante qui peut
                   également s'écrire 1.0e0 ou 0.1e1 ou 0.1E+1 ou 10.0e-1...
- 3.0 + 4 .0       est une expression de valeur 7.0
- 4.3 * 5.0e0      est une expression de valeur 21.5
- 4.0 / 2.0        est une expression de valeur 2.0
- 5.0 / 2.0        est une expression de valeur 2.5 (comparer cf. 3.2.2)
- 2.0 + 3.0 * 4.0  est une expression qui vaut 14.0 (cf. 3.4)
- 2.0 + (3.0 * 4.0) est une expression qui vaut 14.0
- (2.0 + 3.0) * 4.0 est une expression qui vaut 20.0
```

Il est possible d'écrire des valeurs réelles sur l'écran au moyen de *write* (...) (cf. 3.6.5). La notation scientifique est toujours utilisée pour cette écriture (sauf en cas de mention explicite du contraire).

**Exemples**:

```
- write ( 1.0 );       écrit la valeur 1.0 à l'écran sous la forme 0.1e+1
                       (notation scientifique)
- write ( 4.0, 10.0 ); écrit les valeurs 4.0 suivie de 10.0 sur l'écran
- write ( 3.0 + 4.0 ); écrit la valeur de l'expression, à savoir 7.0, sur
                       l'écran

- write ( 'Valeur de pi: ', 3.1416 ); écrit le message Valeur de pi: suivi de la
                                      valeur réelle 3.1416
- const max = 10.0;

var nombre : real; (* pour notre exemple *)
...
    write ( max );             écrit la valeur de max (i. e. 10.0) sur l'écran
    write ( 2.0 * max + 1.0 ); écrit la valeur 21.0 sur l'écran
    write ( max * nombre );    écrit la valeur de max * nombre sur l'écran
```

**Remarque**:

Le type real fait partie des types standard (cf. annexe F).

[ Retour à la table des matières ]

### 3.3.3 **Affectation**

L'affectation s'effectue comme pour les entiers (cf. 3.2.3).

**Exemples:**

```
- nombre := 5.0 + 4.0;           affecte la valeur 9.0 à nb (déclaré cf. 3.3.2)
- nombre := (36.0 / 10.0) * 2.0; affecte la valeur 7.2 à nombre
- nombre := max;                 affecte la valeur de max (cf. 3.3.2) à nombre
```

De manière générale on a:

**nom_de_variable_réelle := expression_de_type_real;**

[ Retour à la table des matières ]

### 3.3.4 **Fonctions prédéfinies**

L'annexe E donne la liste des fonctions prédéfinies. Mentionnons simplement les fonctions suivantes calculant un résultat réel:

```
- abs (valeur)  valeur absolue de valeur, par exemple abs (-3.4) vaut 3.4
- sqr (valeur)  carré de valeur, par exemple sqr (5.0) vaut 25.0
- sqrt (valeur) racine carrée de valeur (>= 0!); exemple: sqrt (4.0) vaut 2.0
- exp (valeur)  nombre e à la puissance valeur
- ln (valeur)   logarithme naturel de valeur
```

[ Retour à la table des matières ]

## 3.4 **Priorité des opérateurs**

En mathématiques une question se pose avec un calcul tel que 2 + 3 * 4: quel est le résultat de ce calcul? Est-ce 20 (calcul 2 + 3 = 5 puis multiplication par 4) ou 14 (ajouter 2 à 3 * 4)?

Le même problème se pose en programmation. Il en existe deux solutions:

- fixer l'ordre d'évaluation (c'est-à-dire de calcul) d'une expression en **utilisant les parenthèses ( et )**. A ce moment la sous-expression entre parenthèses est d'abord calculée. Par exemple:

```
(2 + 3) * 4       2+3 donne 5, multiplié par 4 donne 20
 2 + (3 * 4)      3*4 donne 12, ajouté à 2 donne 14
(3 * (1 + 2)) + 4 1+2 donne 3, multiplié par 3 donne 9, ajouté à 4 donne 13
```

- en l'absence de parenthèses des règles de priorités interviennent. Celles propres à Pascal sont:

a. L'évaluation des fonctions est l'opération la plus prioritaire.
b. Les opérateurs dits multiplicatifs ( * / **div mod**) sont plus prioritaires que les opérateurs dits additifs (+ -).
c. En cas de même priorité l'expression est évaluée de gauche à droite.

**Exemples:**

```
- sin ( x ) * 5.0  calcul de sin ( x ) puis multiplication par 5.0 (règle a)

- 2 + 3 * 4        donne 14 car la sous-expression 3 * 4 est d'abord
                   évaluée (règle b: priorité de * par rapport à + )

- 2 * 3 + 4 div 3  donne 7 car la sous-expression 2 * 3 est d'abord
                   évaluée (règles b puis c), puis la sous-expression
                   4 div 3 l'est (règles b puis c) enfin l'addition des
                   résultats partiels 6 et 1 est effectuée
```

Dans le cas d'une expression comportant des parenthèses et nécessitant l'usage des règles de priorités, les parenthèses sont naturellement considérées en premier.

**Exemple:**

```
- 2 + (3 + 4 * 2) * 5  donne 57 (4 * 2 puis ajouter 3 donne 11,
                       multiplier par 5 puis ajouter 2)
```

[ Retour à la table des matières ]

## 3.5 **Remarques sur les types integer et real**

### 3.5.1 **Conversions de types**

Il est possible d'écrire des expressions formées d'opérandes entières et réelles ainsi que d'opérateurs sur ces deux types. Une telle expression est appelée **expression mixte**. Son évaluation se fait conformément aux règles de priorités en précisant que

- dans une (sous-)expression comportant un opérateur réel, les éventuelles opérandes entières sont converties en opérandes réelles puis le calcul est effectué.

**Exemples:**

```
- 2.0 * 5    5 est converti en 5.0
- 3 - 2.0    3 est converti en 3.0
- 3 / 2      3 et 2 sont convertis en 3.0 et 2.0
```

mais:

```
- (10 div 5) / 2 la conversion n'a lieu qu'après le calcul de 10 div 5
```

D'autre part il est possible d'écrire

**variable_réelle := expression_entière;**

C'est un cas ou l'affectation porte sur des objets de types différents. L'expression entière est d'abord calculée puis sa valeur est convertie en la valeur réelle correspondante et affectée à la variable réelle.

Il existe un autre cas où une affectation entre types différents est autorisée (cf. 7.3.2).

[ Retour à la table des matières ]

### 3.5.2 **Valeurs entières et réelles utilisables en Pascal**

Les valeurs entières utilisables sont celles comprises dans l'intervalle

- maxint..maxint

*maxint* est une constante entière prédéfinie dont la valeur dépend de l'ordinateur et du compilateur Pascal utilisé.
Pour Pascal Macintosh *maxint* vaut 32767 (i. e. $2^{15} - 1$).
Pour Pascal VAX *maxint* vaut 2147483647 (i. e. $2^{31} - 1$).

Les valeurs réelles utilisables dépendent de la représentation en mémoire des nombres réels. Un tel nombre est enregistré sous forme d'une mantisse et d'un exposant, par exemple:

```
0.10012    pour la mantisse
12         pour l'exposant
```

```
donne le nombre 0.10012e12
```

Dans tous les cas il faut se méfier lors de calculs avec les nombres réels, particulièrement si ces nombres sont grands ou petits. En effet une addition telle que

1.0e45 + 1.0e-40 donne 1.0e45 !!!

Ajoutons à cela que les décimales à partir d'une certaine position ne signifient plus rien!

[ Retour à la table des matières ]

## 3.6 **Dialogue programme-utilisateur**

### 3.6.1 **Motivation**

Les programmes vus jusqu'à présent ne sont pas de véritables programmes dans le sens que l'utilisateur ne peut dialoguer avec eux. Or le dialogue entre l'utilisateur et un programme est une partie fondamentale de celui-ci. Plus le dialogue est facile, agréable plus l'utilisateur acceptera d'utiliser le programme.

Sans vouloir approfondir ici la notion de dialogue (cela nous entraînerait beaucoup trop loin), mentionnons simplement qu'il consiste entre autres

pour l'utilisateur:

- à l'introduction des données
- à la commande du programme
- à la compréhension du déroulement des opérations
- à la compréhension des résultats obtenus

pour le programme:

- à la demande des données nécessaires
- à la production de résultats lisibles
- à la quittance des opérations importantes effectuées
- à la mise en garde de l'utilisateur en cas de donnée erronée
- à la mise en garde de l'utilisateur en cas de commande erronée ou dangereuse

Les outils de base pour la programmation d'un dialogue sont les procédures prédéfinies *read, readln* pour la lecture de données (nombres, caractères...) et *write, writeln* pour l'affichage de messages ou de résultats.

[ Retour à la table des matières ]

### 3.6.2 **Lecture de nombres entiers ou réels (read)**

Pour effectuer la lecture d'une valeur entière ou réelle le programme doit

1. Afficher un message à l'utilisateur pour lui indiquer ce qu'il doit faire
2. Suspendre son exécution jusqu'à ce que l'utilisateur ait introduit la valeur
3. Lire la valeur
4. Reprendre son exécution

Ceci se traduit par

```
write ( 'Message pour l''utilisateur' );  (* point 1 *)
read  ( donnee );                         (* points 2, 3 et 4 *)
```

où *donnée* est une variable de type *integer* ou *real*.

**Exemple:**

```
var  rayon : real;     (* rayon d'un cercle *)
...
write ( 'Veuillez donner le rayon du cercle: ' );
read ( rayon );        (* Après cette instruction, rayon a la valeur réelle *)
                       (* tapée au clavier par l'utilisateur *)
```

Pour effectuer la lecture de plusieurs valeurs, il existe l'abréviation suivante:

```
read (v1, v2, ..., vn); qui équivaut à read (v1);
                                       read (v2);
                                       ...
                                       read (vn);
```

Comment cette lecture est-elle effectuée? Considérons l'exemple suivant:

| programme | ce qu'a tapé l'utilisateur |
|---|---|
| **var** longueur : real;<br>largeur : real;<br>hauteur : real;<br>...<br>read ( longueur, largeur, hauteur ); | 142.3 1.0e2 3.5 |

Après l'exécution de read ( *longueur, largeur, hauteur* ):

```
longueur vaut  142.3
largeur  vaut  1.0e2 (c'est-à-dire 100.0)
hauteur  vaut  3.5
```

**Remarque:**

Les nombres doivent être séparés par un (ou plusieurs) espace(s) ou par une (ou plusieurs) marque(s) de fin de ligne. Ceux-ci sont sautés (ignorés) lors de la lecture, sauf ceux présents après le dernier nombre lu (cf. aussi 3.6.3).

[ Retour à la table des matières ]

### 3.6.3 **Passage à la ligne en lecture (readln)**

Dans certains cas il est pratique ou indispensable de ne lire que les premières valeurs tapées par l'utilisateur et d'oublier les autres. Le programme doit donc lire ces premières valeurs (avec *read*) puis ignorer la fin de la ligne où, par hypothèse, se trouvent les autres valeurs. C'est *readln* qui permet d'ignorer la fin de la ligne, y compris la marque de fin de ligne. Exemple:

| programme | ce qu'a tapé l'utilisateur |
|---|---|
| read ( longueur, largeur );<br>readln;<br>read ( hauteur ); | 142.3 1.0e2 3.5<br>12.345 |

Après l'exécution de *read (hauteur)*:

```
longueur vaut  142.3
largeur  vaut  1.0e2
hauteur  vaut  12.345
```

**Remarque:**

Il existe l'abréviation suivante:

```
readln (v1, v2, ..., vn); qui équivaut à read (v1);
                                         read (v2);
                                         ...
                                         read (vn);
                                         readln;
```

L'exemple précédent est donc équivalent à:

```
readln (longueur, largeur);
read (hauteur);
```

**Règle pratique**

Pour éviter toute surprise, chaque fois qu'une donnée tapée par l'utilisateur se termine par une marque de fin de ligne (cf. 10.4.1), cette marque doit être sautée (éliminée) par l'utilisation de *readln*.

[ Retour à la table des matières ]

### 3.6.4 **Ecriture de messages à l'écran (write)**

Un programme bien fait comporte toujours des ordres d'affichage de messages à l'écran. Ces messages indiquent à l'utilisateur qu'il faut introduire une donnée, que telle opération est terminée...

Ces messages sont en fait des constantes du type "chaîne de caractères" (cf. 8.2). Un message doit donc s'écrire entre apostrophes. Attention: toute apostrophe faisant partie du message doit être dédoublée.

**Exemples de messages:**

'ceci est un message'
'ceci est un autre message'
'ATTENTION: une apostrophe dans un message doit etre dedoublee'
'Ce dedoublement n''a pas ete oublie ici'

L'affichage d'un message est maintenant facile:

**write ('Affichage de ce message');**

**Remarque:**

Il est important que:

a) tout programme "s'annonce" dès le début de son exécution par un message àl'utilisateur. Celui-ci obtient donc une confirmation du début d'exécution du programme.

b) tout programme fasse "patienter" l'utilisateur par un message lorsqu'il effectue des opérations nécessitant un temps relativement long.

[ Retour à la table des matières ]

### 3.6.5 **Ecriture de valeurs entières ou réelles (write)**

La procédure *write* s'utilise aussi pour écrire des valeurs entières ou réelles (cf. 3.2.2 et 3.3.2). Il faut ici donner les précisions suivantes:

1. L'abréviation suivante est agréable à utiliser:

```
write (e1, e2, ..., en); qui équivaut à write (e1);
                                        write (e2);
                                        ...
                                        write (en);
```

2. Toute valeur entière est écrite sur un certain nombre de positions qui dépend du compilateur utilisé (Pascal Macintosh: 8 positions, Pascal VAX: 10). Cela signifie que l'instruction

write ( 'Le resultat vaut:', 56 );

```
aura l'effet suivant:    Le resultat vaut:      56
```

avec 6 (sur VAX: 8) espaces entre le caractère : et le nombre 56.
De même toute valeur réelle est écrite en notation scientifique sur 10 positions avec Pascal Macintosh, 12 avec Pascal VAX.

3. Il est possible de modifier ce nombre de positions de la manière suivante:

```
- write ( 'Le resultat vaut:', 56 : 4 ); force l'écriture de 56 sur 4 positions.
- write ( 12.345 : 7 );                  force l'écriture de 12.345 en notation
                                         scientifique sur 7 positions.
- write ( 12.345 : 7 : 4 );              force l'écriture de 12.345 en notation
                                         décimale sur 7 positions avec 4 chiffres
                                         après la virgule.
```

**Remarque:**

Le(s) signe(s) et le point décimal comptent chacun pour une position.

[ Retour à la table des matières ]

### 3.6.6 **Passage à la ligne en écriture (writeln)**

Le passage à la ligne suivante s'effectue au moyen de la procédure prédéfinie *writeln*.
Par exemple:

```
write ( 'Le premier resultat est:', 17 : 4 );
writeln;
write ( 'Le deuxieme vaut:', 45 );
writeln;
```

```
donne   Le premier resultat est: 17
        Le deuxieme vaut: 45
```

avec la prochaine écriture au début de la ligne suivante.

Il est possible d'abréger ainsi:

```
writeln (e1, e2, ..., en); qui équivaut à write (e1, e2, ..., en);
                                           writeln;
```

[ Retour à la table des matières ]

**CHAPITRE 4**

# L' ITERATION, LE CHOIX, LES AUTRES TYPES STANDARD

---

## 4.1 **Itération: la boucle for**

### 4.1.1 **Motivation**

Dans tout programme sérieux certaines actions sont répétées plusieurs fois. Par exemple le programmeur veut créer le dessin suivant formé de 10 triangles:

Le lecteur remarque immédiatement que cela revient à dessiner 10 fois le même triangle.

On appelle **itération** une répétition d'un groupe d'instructions. Notre exemple comporte donc 10 itérations.

[ Retour à la table des matières ]

### 4.1.2 **La boucle for**

La réalisation d'itérations se fait en utilisant une instruction appelée communément **boucle** (il y a trois boucles différentes en Pascal). Le fait de connaître **à l'avance** le nombre d'itérations conduit à choisir l'instruction **for**. On écrit:

```
for i := 1 to 10 do       (* i est une variable entière *)
    " dessiner le triangle "
```

Cette instruction for va s'exécuter de la manière suivante: la variable de boucle *i* va prendre successivement les valeurs 1, 2, 3... 9 et 10, et pour chacune de ces valeurs l'action (l'instruction) "dessiner le triangle" qui suit le mot réservé **do** va s'effectuer.

Un premier essai de programme complet est alors

```
program triangle_10_premier_essai (input, output);
(* Auteur... *)

   const nombre_de_triangles = 10;  (* nombre de triangles à dessiner *)
         x_init = 10;               (* abscisse du point de dessin initial *)
         y_init = 30;               (* ordonnée du point de dessin initial *)

   var   i : integer;       (* variable de boucle *)

   begin (* triangle_10_premier_essai *)

     showdrawing;

     moveto (x_init, y_init); (* déplacement au point (x_init, y_init) *)
     for i := 1 to nombre_de_triangles do (* dessiner les 10 triangles *)
     begin
         line (40, 0);        (* dessin de la base *)
         line (- 20, - 20);   (* dessin du côté à droite *)
```

```
            line (- 20, 20);    (* dessin du dernier côté *)
        end;

    end.
```

Une lecture attentive de ce programme fait apparaître un problème: les triangles seront tous dessinés au même endroit. Il faut donc changer de point initial avant chaque dessin. On remarque que pour le n$^{ème}$ triangle, ce point a les coordonnées

( x_init + (n - 1) * 40, y_init )

D'autre part les nombres 20 et 40 présents sont à remplacer par une (des) constante(s) déclarée(s)!

Le programme modifié en conséquence devient alors correct:

```
program triangle_10_correct (input, output);
(* Auteur... *)

   const   nombre_de_triangles = 10;    (* nombre de triangles à dessiner *)
           x_init = 10;                 (* abscisse du point de dessin initial *)
           y_init = 30;                 (* ordonnée du point de dessin initial *)
           base = 40;                   (* longueur de la base d'un triangle *)

   var     i : integer;     (* variable de boucle *)

begin (* triangle_10_correct *)

   showdrawing;

   for i := 1 to nombre_de_triangles do        (* dessiner les 10 triangles *)
   begin
      moveto (x_init + (i - 1) * base, y_init); (* déplacement au point suivant *)
      line ( base, 0 );                         (* dessin de la base *)
      line (- base div 2, - base div 2);       (* dessin du côté à droite *)
      line (- base div 2, base div 2);         (* dessin du dernier côté *)
   end;

end.
```

La forme générale de l'instruction **for** est:

**for variable_de_boucle := expression_1 to expression_2 do**
**instruct*ion;***

avec variable_de_boucle, expression_1 et expression_2 de **type scalaire** (cf. annexe F).

**Remarques importantes:**

1. Après la fin de l'exécution de la boucle, la variable de boucle (appelée aussi variable de contrôle) **n'a pas de valeur définie.**

2. **Il n'est pas possible** de changer la valeur de la variable de contrôle dans la boucle.

3. Si *expression_1 > expression_2* initialement, **la boucle n'est pas effectuée.**

4. Si la valeur de *expression_1* ou *expression_2* est modifiée dans la boucle, le nombre d'itérations **ne change pas !**

5. L'instruction **begin ... end** est appelée **instruction composée**. Elle a pour but d'envelopper un groupe d'instructions et de permettre ainsi leur répétition (cf. ci-dessus), leur exécution ou non (cf. 4.2.2)... Plus précisément elle permet à ce groupe d'être considéré comme une seule instruction, donc d'être utilisé partout où une instruction peut se trouver.

6. Il est possible d'écrire **for** nom_variable := e1 **downto** e2 **do** instruction; avec *e1 > e2* . Dans ce cas *nom_variable* décroît de *e1* à *e2*.

[ Retour à la table des matières ]

## 4.2 **L'instruction if**

### 4.2.1 **Motivation**

Les premiers programmes présentés s'exécutaient en **séquence**, c'est-à-dire une instruction après l'autre en suivant leur ordre d'écriture dans le corps du programme.

Lors de la présentation du concept d'itération (cf. 4.1.1), un bloc d'instructions pouvait être introduit dans le corps de la boucle, ce qui provoquait sa **répétition** un nombre de fois fixé par les bornes inférieure et supérieure de l'intervalle de variation de la variable de contrôle.

Nous allons voir comment programmer un **choix,** c'est-à-dire comment faire pour exécuter ou ne pas exécuter un bloc d'instructions. Cette possibilité nous est donnée par l'instruction **if.**

[ Retour à la table des matières ]

### 4.2.2 **L'instruction if sans alternative**

Supposons que nous voulions maintenant écrire un programme qui affiche tous les nombres impairs entre 5 et 23. Soit *nb_courant* le nombre examiné; déterminer s'il est pair ou impair peut être réalisé en considérant le reste de la division (entière) par 2. En effet **si** *nb_courant* **mod** 2 vaut 0 **alors** nb_courant est pair; **si** *nb_courant* **mod** 2 vaut 1 **alors** *nb_courant* est impair.Il est donc possible de n'afficher que les nombres impairs en effectuant une **distinction de cas**, c'est-à-dire que nous ne voulons exécuter une instruction d'écriture que **si** une **condition** est réalisée. Cela s'écrit:

```
if nb_courant mod 2 = 1 then (* nb_courant est-il impair? *)
     writeln ( 'Le nombre ',nb_courant : 3,' est impair' );
```

Le message Le nombre ... impair n'est affiché que **si** la condition *nb_courant* mod *2 = 1* est **vraie,** i.e. lorsque *nb_courant* est impair.

La suite d'instructions effectuant l'affichage des nombres impairs entre 5 et 23 est alors:

```
const borne_inf = 5;
      borne_sup = 23;
   ...
 var  nb_courant : integer;
   ...
begin (* ... *)
  ...
  for nb_courant := borne_inf to borne_sup do
      if nb_courant mod 2 = 1 then
          writeln ( 'Le nombre ',nb_courant : 3,' est impair' );
 ...
end.
```

L'instruction **if** (appelée aussi énoncé conditionnel) s'utilise donc chaque fois qu'un **choix** entre différentes options doit être

programmé.

Sa forme générale est:

***if condition then instruction;***

La **condition** est en fait une expression dont le résultat d'évaluation est **soit vrai soit faux**. Une telle expression est appelée **expression booléenne** (cf. 4.3.1).

**Exemples d'expressions booléennes**:

```
Soient expression_1 et expression_2 deux expressions de même type:

  expression_1 =  expression_2  34 + 5 = 56 - 17 est vraie
  expression_1 <> expression_2  34 + 5 <> 56 - 17 est fausse
  expression_1 <= expression_2  1 <= 5 est vraie, de même que 1 <= 1
  expression_1 <  expression_2  3 < 5*5 est vraie
  expression_1 >= expression_2  5 >= 1 est vraie, de même que 1 >= 1
  expression_1 >  expression_2  1 > 5 est fausse
```

Les symboles = <> <= < >= > sont appelés **opérateurs de comparaison.**

**Remarques:**

1. Si plusieurs instructions doivent être exécutées conditionnellement, il faut l'instruction composée **begin** ... **end** (cf. 4.1.2).
   Exemple :

   **if** nb_courant **mod** 2 = 1 **then**
   **begin**
   writeln( 'Le nombre ',nb_courant : 3,' est impair');
   writeln( 'Son carré ',sqr(nb_courant) : 4, ' aussi');
   **end**;

2. Plusieurs conditions peuvent être combinées en une seule, par exemple :

   **if** (nb_courant **mod** 2 = 1) **and** (nb_courant > 20) **then ...**

Le mot réservé **and** est un opérateur booléen réalisant la fonction logique **et**. Nous verrons cela lors de la présentation du type boolean (cf. 4.3).

[ Retour à la table des matières ]

### 4.2.3 **L'instruction if avec alternative**

Reprenons le cas des nombres pairs et impairs et modifions cet exemple de manière à afficher un message si *nb_courant* est impair et un autre message s'il est pair. Cela s'écrit:

```
if nb_courant mod 2 = 1 then                (* nb_courant est-il impair? *)
   writeln( 'Le nombre ',nb_courant : 3,' est impair') (* il est impair *)
else
   writeln( 'Le nombre ',nb_courant : 3,' est pair');  (* il est pair *)
```

La partie **else** est exécutée si la condition est fausse, ici si *nb_courant* est pair.

La forme générale est:

***if condition then instruction_1 else instruction_2;***

**ATTENTION**: C'est une erreur syntaxique de mettre un point-virgule après l'instruction précédant le mot réservé **else**!

D'où la règle pratique: "**On met toujours un point-virgule après une instruction sauf avant un** else".

**Remarque**:

Quel est l'effet de la partie de programme suivante:

```
if condition_1 then
  if condition_2 then
       instruction_1
  else instruction_2;
```

Cette situation est ambiguë. Par convention la partie **else** correspond à l'instruction **if la plus proche**, ici **if** condition_2 **then** ...

Le contraire s'obtient à l'aide d'une instruction composée:

```
if condition_1 then
begin
  if condition_2 then
       instruction_1;
end
else instruction_2;
```

[ Retour à la table des matières ]

## 4.3 **Le type boolean**

### 4.3.1 **Généralités**

*Boolean* est un **identificateur prédéfini.** Il signifie "un objet (constante, variable, expression...) de type *boolean* aura la valeur vrai ou faux et rien d'autre". Le type *boolean* fait partie des types standard et des types scalaires (cf. annexe F).

Les **constantes** booléennes sont:

```
false    true
```

qui représentent respectivement les valeurs **faux** et **vrai.**

Les **opérations** possibles sur les valeurs de type boolean sont:

```
      and   or   not   =   <>   <=   <  >=  >

and  représente la fonction logique et
or   représente la fonction logique ou (non exclusif)
not  représente la fonction logique non
```

Les opérateurs de comparaison prendront un sens lorsque nous aurons précisé que, par définition, le type boolean est **ordonné** et que **false** < **true.**

Les **expressions booléennes** s'écrivent comme le montrent les exemples qui suivent:

```
var trouve,                  (* pour cet exemple *)
    present : boolean;
trouve and present       expression vraie si trouve et present sont vrais
```

```
not trouve              expression vraie si trouve est faux
trouve or present       expression vraie si trouve ou present est vrai
not (trouve and present) expression vraie si trouve ou present est faux
not trouve or not presentexpression équivalente à la précédente
...
```

Les parenthèses servent à préciser l'ordre d'évaluation d'une expression. De plus la priorité des opérateurs booléens est (dans l'ordre décroissant):

```
not       (le plus prioritaire)
and
or
```

opérateurs de comparaison (les moins prioritaires)

A titre de résumé, les opérations permettant la construction d'une expression d'un type quelconque sont, dans **l'ordre décroissant** des priorités:

évaluation d'une fonction (opération la plus prioritaire)
**not**
**and**
**or**
opérateurs multiplicatifs ( * / **div mod** )
opérateurs additifs ( + - )
opérateurs de comparaison ( = <> <= < >= > )

**Remarque**:

Nous avons vu qu'une condition (cf. 4.2.2 et 4.2.3) est en fait une expression booléenne. La forme générale de l'instruction **if** peut donc s'écrire:

***if expression_booléenne then instruction_1***
***else instruction_2;***

[ Retour à la table des matières ]

### 4.3.2 **Affectation**

L'affectation s'effectue comme habituellement:

```
var trouve  : boolean;          (* pour cet exemple *)
    present : boolean;
    nb_courant : integer;
...
   trouve  := false;
   present := trouve;
   trouve  := nb_courant = 25;   (* trouve vaut alors vrai si nb_courant *)
                                 (* vaut 25, faux sinon *)
```

Donc de manière générale:

***nom_de_variable_booléenne := expression_booléenne;***

[ Retour à la table des matières ]

### 4.3.3 **Fonctions prédéfinies**

Il y a principalement 3 fonctions prédéfinies à résultat de type boolean:

```
odd ( expr_entiere )      vraie si expr_entiere représente un nombre impair,
                          fausse sinon
eof (...) et eoln (...)   cf. 10.4.2
```

[ Retour à la table des matières ]

### 4.3.4 **Entrées-sorties**

Le Pascal standard n'admet aucune entrée-sortie sur des valeurs du type *boolean*. Cependant plusieurs compilateurs fournissent cette extension, comme par exemple Pascal Macintosh et Pascal VAX. Dans ces deux cas la sémantique est la suivante:

- la lecture et l'écriture d'une valeur booléenne s'effectuent comme la lecture et l'écriture d'un nombre entier.

**Exemples:**

```
var trouve : boolean;      (* pour notre exemple *)
...
   read  ( trouve );
   write ( false );         (* écrit la constante false sur le nombre minimum *)
                            (* de positions, à savoir 5 *)
   write ( trouve : 10 );   (* si trouve a la valeur true on obtient l'écriture *)
                            (* de six espaces suivis du mot true *)
```

[ Retour à la table des matières ]

### 4.4 **Itération: la boucle while**

La première construction permettant de répéter l'exécution d'un groupe d'instructions est l'instruction **for.** L'emploi de cette construction est possible (et recommandé) lorsque le nombre de répétitions est **connu à l'avance.**

L'instruction **while** permet de répéter l'exécution d'un groupe d'instructions un nombre de fois **non connu à l'avance.** Cette répétition s'effectue tant qu'une condition (expression booléenne) **est vraie.**

**Exemple**:

```
const nombre_final = 0;          (* permet de terminer le programme lorsque *)
                                 (* l'utilisateur le donne *)
var nombre : integer;            (* nombre donné par l'utilisateur *)

begin (* ... *)
   nombre := 1;                   (* initialisation à ne pas oublier! *)
   while nombre <> nombre_final do   (* traiter les nombres de l'utilisateur *)
   begin
      write ( 'Donnez la valeur suivante: ' );
      readln ( nombre );
      if nombre <> nombre_final then
         ... (* traitement du nombre *)
   end;
   ... (* instructions suivantes *)
```

L'instruction qui suit le mot réservé **do** (ici l'instruction composée **begin** ... **end**) est répétée tant que la condition *nombre <>nombre_final* est vraie, i.e. tant que *nombre* est différent de 0.

Dans notre exemple, l'utilisation de l'instruction composée permet de répéter l'exécution des instructions "enveloppées" par **begin** et **end.**

La forme générale de l'instruction **while** est:

***while expression_booléenne do instruction;***

**Remarques**:

1. Il faut s'assurer que l'expression booléenne devient fausse après un nombre fini d'itérations, faute de quoi le programme *exécute l'instruction* ***while*** *indéfiniment.* On dit alors que le programme "**boucle"**, que la boucle est **infinie.**

2. Si l'expression booléenne est **initialement fausse,** l'instruction **while** n'est pas effectuée.

[ Retour à la table des matières ]

## 4.5 **Le type char**

### 4.5.1 **Généralités**

Char est un **identificateur prédéfini.** Il signifie "un objet (constante, variable, expression...) de type *char* contiendra un caractère et rien d'autre". Le type char fait partie des types standard et des types scalaires (cf. annexe F).

Les **constantes** de type *char* sont des caractères écrits entre apostrophes comme par exemple

```
'A'   'a'   '1'   '?'   '/'   etc.
```

Ces constantes sont ordonnées selon un code, unique pour chaque caractère. Il existe plusieurs codes utilisés en informatique (EBCDIC ...), le plus courant et le **seul utilisé dans ce cours** étant le code **ASCII** (cf. annexe A).

Chaque **caractère imprimable** (caractère situés entre l'espace et le tilde ~ y compris) du code ASCII peut être écrit entre apostrophes. Les autres peuvent être utilisés par le biais de la fonction chr (cf. 4.5.3).

Les **opérations** possibles sur les valeurs de type char sont les comparaisons:

```
=   <>   <=   <   >=   >
```

Les **expressions** de type *char* se limitent aux constantes, variables et fonctions.

**Remarques:**

1. La constante "caractère apostrophe" s'écrit "" (quatre apostrophes accolées).

2. Quel que soit le code utilisé, on a toujours

```
'A' < 'B' < 'C' < ... < 'Z'
'a' < 'b' < 'c' < ... < 'z'
'0' < '1' < '2' < ... < '9'
```

même si d'autres caractères sont intercalés entre deux lettres ou deux chiffres (!) ce qui n'est pas le cas en **ASCII.**

[ Retour à la table des matières ]

### 4.5.2 **Affectation**

L'affectation s'effectue comme habituellement:

```
var lettre : char; (* pour notre exemple *)
    lettre_courante : char;
```

```
...
   lettre_courante := ' '; (* caractère espace *)
   lettre := lettre_courante;
```

Donc de manière générale:

***nom_de_variable_de_type_char := expr_de_type_char;***

[ Retour à la table des matières ]

### 4.5.3 **Fonctions prédéfinies**

Il y a 3 fonctions prédéfinies à résultat de type *char*:

| | |
|---|---|
| `succ ( expr_de_type_char )` | `fournit le caractère suivant celui indiqué par` *`expr_de_type_char`* `(erreur s'il n'existe pas)` |
| `pred ( expr_de_type_char )` | `fournit le caractère précédant celui indiqué par` *`expr_de_type_char`* `(erreur s'il n'existe pas)` |
| `chr ( expr_entière )` | `fournit le caractère dont le code est` *`expr_entière`* `(erreur s'il n'existe pas)` |

**Exemples:**

```
succ ( 'A' ) vaut 'B'
pred ( 'B' ) vaut 'A'
chr  ( 32  ) vaut ' ' (* le caractère espace *)
chr  ( 65  ) vaut 'A'
```

**Remarque:**

Il existe la fonction inverse de chr (...):

| | |
|---|---|
| `ord ( expr_de_type_char )` | `fournit le code (valeur entière) du caractère indiqué par` *`expr_de_type_char`* |

```
Par exemple: ord ('A' ) vaut 65

On a donc: ord ( chr ( 74  ) ) qui vaut 74
           chr ( ord ( '>' ) ) qui vaut '>'
```

[ Retour à la table des matières ]

### 4.5.4 **Type char et instruction for**

Des expressions de type *char* peuvent être utilisées dans une boucle **for** (cf. 4.1.2).

Par exemple:

```
for lettre := 'a' to 'z' do ...;
```

[ Retour à la table des matières ]

### 4.5.5 **Entrées-sorties**

Les instructions *read* et *readln* permettent de lire des caractères:

```
var lettre,     (* pour notre exemple *)
```

```
      caractere : char;
   ...
   read ( lettre );       (* lit un caractère *)
   readln ( caractere );  (* lit un caractère et saute toute la fin de la ligne *)
```

L'écriture de caractères s'effectue par *write* et *writeln:*

```
   write ( lettre );       (* écrit le caractère contenu dans lettre sur une position
*)
   writeln ( caractere );  (* écrit le caractère contenu dans caractere sur une *)
                           (* position et termine la ligne *)

   write ( caractere : expr_entière ); (* écrit le caractère contenu dans caractere
*)
                                       (* sur expr_entière positions. Les positions
*)
                                       (* supplémentaires sont des espaces *)
```

**Remarque**

Il ne faut pas essayer de lire la marque de fin de ligne (cf. 10.4.2).

[ Retour à la table des matières ]

## 4.6 **Déclaration des constantes et des variables**

### 4.6.1 **Déclaration des constantes**

Les constantes se déclarent dans la zone (de la partie déclarative) commençant par le mot réservé **const,** avant les variables s'il y en a. Chaque déclaration a la forme

***identificateur_de_constante = valeur_constante;***

où *valeur_constante* peut être un nombre entier ou réel, une valeur booléenne, un caractère, une chaîne de caractères ou un identificateur d'une constante déjà déclarée.

**Exemples**:

```
   const  pi = 3.14159;          (* constante réelle *)
          zero = 0;              (* constante entière *)
          vrai = true;           (* utilité??? *)
          espace = ' ';          (* constante caractère *)
          bonjour = 'Bonjour!';  (* constante chaîne de caractères *)
          nul = zero;            (* constante entière *)
```

**Remarque**

Il ne faut pas écrire le mot **const** (dans une partie déclarative) si aucune constante n'est déclarée ensuite!

[ Retour à la table des matières ]

### 4.6.2 **Déclaration des variables**

Les variables se déclarent dans la zone (de la partie déclarative) commençant par le mot réservé **var,** après les constantes s'il y en a. Chaque déclaration a la forme

***identificateur_de_variable : type;***

où type peut être n'importe quel type (cf. annexe F).

**Exemples:**

```
var  cote   : integer;  (* variable entière   *)
     racine : real;     (* variable réelle    *)
     trouve : boolean;  (* variable booléenne *)
     lettre : char;     (* variable caractère *)
```

**Remarque**

Il ne faut pas écrire le mot **var** (dans une partie déclarative) si aucune variable n'est déclarée ensuite!

[ Retour à la table des matières ]

**CHAPITRE 8**

# LES ENSEMBLES ET LES CHAINES DE CARACTERES

## 8.1 **Les types ensemble**

### 8.1.1 **Motivation**

Les types ensemble permettent de représenter des ensembles au sens mathématique du terme. Ces ensembles sont utiles chaque fois qu'une donnée est formée de plusieurs éléments tous de même type et que des **groupes** de ces éléments sont manipulés. De telles données sont utilisées dans des problèmes comme:

- l'analyse lexicale
- la manipulation des bits d'un mot machine
- plus généralement chaque fois que le test suivant doit être programmé (test d'appartenance): une valeur V se trouve-t-elle ou non parmi un groupe donné?

...

[ Retour à la table des matières ]

### 8.1.2 **Généralités**

Un ensemble est une collection (non ordonnée) d'éléments (**membres**) tous de même type appelé type de base. Le type de base peut être *char*, *boolean*, un type énuméré ou un type intervalle.

Exemples:

```
- set of char;                   (* ensembles de caractères *)
- set of t_jours_de_la_semaine;  (* ensembles de jours (cf. 7.1.2) *)
- set of t_chiffre;              (* ensembles d'entiers compris entre 0 et 9 *)
                                  (* (cf. 7.3.1) *)
```

Les types ensemble se déclarent ainsi:

```
type  t_signes_de_ponctuation = set of char;

      t_nos_des_jours = 1..365;                (* type intervalle, cf. 7.3 *)
      t_jours_feries = set of t_nos_des_jours; (* attention à la cardinalité des
*)
                                               (* ensembles, cf. remarque 2 *)
                                               (* ci-après *)
```

et les variables de ces types:

```
var  signes : t_signes_de_ponctuation;

      jours_feries_annee_1986 : t_jours_feries;
```

Les **constantes** d'un type ensemble sont des (sous-)ensembles écrits de la manière suivante:

a) suite de constantes et/ou de variables du type de base séparées par des virgules:

```
- [1, 8, 2, 5]        ensemble des entiers 1, 2, 5 et 8
- [3]                 ensemble formé du seul entier 3
- [1, i, j+1]         ensemble formé de l'entier 1 et des valeurs
                      (nombres entiers) de i et j+1
- ['a', 'A']          ensemble formé des caractères 'a' et 'A'
- [samedi, dimanche]  ensemble des constantes énumérées samedi et
                      dimanche (cf. 7.1.2)
```

b) suite d'intervalles formés sur le type de base et séparés par des virgules.
Remarquez qu'il est possible d'utiliser des **variables** comme bornes des intervalles:

```
- [3..5]               ensemble des entiers 3, 4 et 5
- ['a'..'z', 'A'..'Z'] ensemble des lettres majuscules et minuscules
- [1..j]               ensemble des entiers de 1 à j ou ensemble vide
                       si j < 1
```

c) combinaison de a) et b):

```
- [1, 3, 6..10, 12]
- [i, 1..j-1]
```

d) ensemble vide noté []

Les **opérations possibles** (en plus de l'affectation et du passage en paramètre) sur les valeurs d'un type ensemble sont:

- la réunion (notée +), l'intersection (notée *) et la différence (notée -) de deux ensembles
- l'égalité (notée =), l'inégalité (notée <>), l'inclusion (notée <= ou >=)

**Exemples:**

```
var   signes_1,
      signes_2 : t_signes_de_ponctuation; (* voir plus haut *)

- signes_1 + signes_2   ensemble formé de tous les éléments de signes_1 et
                        signes_2
- signes_1 * signes_2   ensemble formé des éléments communs à signes_1 et
                        signes_2
- signes_1 - signes_2   ensemble formé des éléments de signes_1
                        n'appartenant pas à signes_2
- signes_1 = signes_2   expression vraie si les deux ensembles sont égaux
- signes_1 <> signes_2  expression vraie si les deux ensembles ne sont pas
                        égaux
- signes_1 <= signes_2  expression vraie si signes_1 est contenu dans
                        signes_2
- signes_1 >= signes_2  expression vraie si signes_1 contient signes_2
```

En plus des constantes et des variables, les expressions sont formées de réunions, d'intersections et de différences d'ensembles de même type.

**Remarques:**

1. Il n'existe pas de fonctions (même prédéfinies) à résultat d'un type ensemble et aucune entrée-sortie n'est possible pour un ensemble.

2. Certains compilateurs Pascal imposent une limite sur la taille maximale des ensembles. Si elle existe cette limite est généralement petite! Par exemple la limite est de 256 pour Pascal VAX (pas de limite pour Pascal Macintosh).

3. Les types ensemble font partie des types structurés (cf. annexe F).

4. Notons pour les puristes qu'une "constante" comme [1, 2, i, j] s'appelle en fait un agrégat.

[ Retour à la table des matières ]

### 8.1.3 **Affectation**

L'affectation se fait de manière habituelle entre **ensembles du même type**:

```
signe_1 := signe_2;      (* cf. 8.1.2 *)
```

[ Retour à la table des matières ]

### 8.1.4 **Test d'appartenance**

Le test d'appartenance est une construction très utilisée en relation avec les ensembles. Au moyen de l'opérateur **in** le programmeur peut savoir si une valeur (du type de base) appartient ou non à un ensemble.

**Exemples:**

| | |
|---|---|
| `- 3 in [1..i]` | expression booléenne vraie si 3 appartient à l'ensemble [1..i] (donc si *i* >= 3 ) |
| `- 56 in jours_feries_annee_1986` | expression booléenne vraie si le 56ème jour de 1986 est férié (cf. 8.1.2) |
| `- ch in signes_1 + signes_2` | expression booléenne vraie si le caractère contenu dans *ch* appartient à l'ensemble formé de la réunion *signes_1* + *signes_2* (cf. 8.1.2) |

**Remarque:**

L'opérateur **in** a la même priorité que les opérateurs de comparaison.

[ Retour à la table des matières ]

## 8.2 **Les types "chaîne de caractères"**

### 8.2.1 **Motivation**

Notre monde actuel est ainsi fait que (presque) tout élément le composant possède un (ou plusieurs) nom(s) permettant de l'identifier. Ce ou ces noms peuvent bien entendu être complétés par un ou plusieurs numéros mais il est rare que seul un code

numérique soit utilisé. Quel sens aurait en effet la phrase "11 passe à 9 puis à 4" ? Notre imagination nous suggère plusieurs possibilités: une action de football, le nombre de pourcents de l'inflation au cours du temps, l'évolution des taux hypothécaires (!) ...

Cet état de fait implique que tout traitement informatique comportera des manipulations de tels noms appelés **chaînes de caractères**. Formellement une chaîne de caractères est une suite de zéro, un ou plusieurs caractères accolés.

**Remarques:**

1. La chaîne formée d'aucun caractère s'appelle la **chaîne vide**.
2. Une constante caractère peut être vue comme une constante chaîne de caractères de longueur 1!

En Pascal le traitement des chaînes de caractères est un point où les versions du langage sont toutes différentes. Cependant plusieurs d'entre-elles offrent des opérations de manipulation similaires, agréables à utiliser. Parmi ces versions mentionnons Pascal Macintosh, Pascal VAX, Turbo Pascal...

Nous allons exposer ici la situation de Pascal Macintosh.

[ Retour à la table des matières ]

### 8.2.2 **Généralités (Pascal Macintosh)**

Une chaîne de caractères est une suite de caractères définie par

- une **taille** maximale fixe
- une **longueur courante** variable jamais supérieure à la taille
- les caractères présents dans la chaîne

Une chaîne de caractères se déclare au moyen du mot réservé **string.**

**Exemples**:

```
- string [ 10 ]   (* chaîne de taille égale à 10 *)
- string [ 1 ]    (* la plus petite chaîne possible *)
- string [ 255 ]  (* la plus grande chaîne possible *)
```

```
Les types chaîne se déclarent ainsi:

   const  max_car_nom = 30;    (* il est rare d'avoir un nom de 30 lettres. *)
          max_car_prenom = 15; (* pour le prénom principal seulement. *)
          max_ligne = 80;      (* longueur maximale d'une ligne de texte. *)

   type  t_nom = string [ max_car_nom ]; (* max_car_nom est la taille*)
         t_prenom = string [ max_car_prenom ];
         t_ligne = string  [ max_ligne ];
         t_longue_ligne = string;        (* équivaut ici à string [ 255] *)

Pour les variables de ces types:

   var   nom_de_famille : t_nom;
         prenom_principal : t_prenom;
         ligne_courante : t_ligne;
```

Les **constantes** d'un type chaîne sont écrites entre apostrophes:

```
'Abcd'  de taille et de longueur courante 4
''      chaîne vide (taille = longueur courante = 0)
'x'     de taille et de longueur courante égales à 1. C'est aussi une constante
        caractère!
```

Les **opérations** possibles (en plus de l'affectation et du passage en paramètre) sur les chaînes de caractères sont:

```
= < <= <> > >= entre chaînes de n'importe quel type.
```

Les **expressions** sont réduites aux constantes, variables et fonctions prédéfinies à résultat d'un type chaîne.

**Remarque:**

Les types chaîne de caractères font partie des types structurés (cf. annexe F).

[ Retour à la table des matières ]

### 8.2.3 **Accès aux éléments d'une chaîne de caractères**

Il est possible d'accéder aux caractères composant une chaîne par la même notation que celle utilisée pour les éléments d'un tableau. Seul l'accès aux caractères présents est permis.

**Exemple:**

```
var nom_de_famille : t_nom; (* cf. 8.2.2 *)
...
nom_de_famille := 'Dupont'; (* cf. 8.2.4 *)

alors

   nom_de_famille [ 1 ] vaut 'D'
   nom_de_famille [ 2 ] vaut 'u'
   ...
   nom_de_famille [ 6 ] vaut 't'

   nom_de_famille [ 7 ] n'existe pas.    Un accès en lecture ou en écriture
                                         provoque une erreur car 7 est supérieur
                                         à la longueur courante (égale à 6)

   nom_de_famille [ 0 ] n'existe jamais. Tout accès provoque une erreur
```

[ Retour à la table des matières ]

### 8.2.4 **Affectation**

L'affectation est possible entre chaînes de **n'importe quels types** (!) et de longueur quelconque, à condition que la taille de la chaîne affectée ne soit pas dépassée!

**Exemples:**

```
var nom_de_famille : t_nom;  (* cf. 8.2.2 *)
```

```
...
nom_de_famille := 'Dupont';  (* après affectation, nom_de_famille est de *)
                             (* longueur 6 *)
nom_de_famille := '';        (* après affectation, nom_de_famille est de *)
                             (* longueur 0 *)
```

Il est faux d'écrire nom_de_famille := '01234567890123456789012345678 9abcde'; car la constante est de longueur 35, longueur supérieure à la taille de *nom_de_famille*.

[ Retour à la table des matières ]

### 8.2.5 **Procédures et fonctions prédéfinies**

Jusqu'à présent le maniement des chaînes est rudimentaire. Il va être grandement amélioré par les procédures et fonctions de manipulation, qui sont prédéfinies.

[ Retour à la table des matières ]

#### 8.2.5.1 **Fonctions prédéfinies**

Soient *s*, *s1*, *s2*, ..., sn, sub des chaînes de caractères et *indice*, *nb* des nombres entiers. On a

`length ( s )` donne la longueur courante de *s*

`concat ( s1, s2, ... sn )` donne la concaténation des chaînes *si*. Il est possible de concaténer également des caractères!

`pos ( sub, s )` donne la position du premier caractère de *sub* dans *s* si *sub* est une sous-chaîne de *s*, 0 sinon

`copy ( s, indice, nb )` donne une copie de la sous-chaîne de longueur *nb* commençant au caractère *s [ indice ]*

`omit ( s, indice, nb )` donne une copie de la chaîne formée des caractères *s [ 1 ], s [ 2 ], ... s [ indice-1 ], s [ indice+nb ], s [ indice+nb+1 ], s [ indice+nb+2 ], ..., s [ length ( s ) ]*

`include ( sub, s, indice )` inclut la chaîne sub dans une copie de *s* à partir du caractère *s [ indice ]* et donne le tout comme résultat

**Exemples:**

```
const   max = 10;

type    t_chaine = string [ max ];

var      s,
         sub :t_chaine;
begin
```

```
      s := 'ABCD';
      sub := '123';

alors

      length ( s )           donne 4
      length ( sub )         donne 3

      concat ( s, sub )      donne la chaîne ABCD123
      concat ( s, ' ', sub ) donne la chaîne ABCD 123

      pos ( 'BC', s )        donne 2
      pos ( sub, s )         donne 0
      pos ( 'BC', 'ABCABC')  donne 2

      copy ( s, 3, 2 )       donne la chaîne CD
      copy ( s, 2, 3)        donne la chaîne BCD
      copy ( s, indice, 0)   correct pour tout valeur indice entière! Donne la
      chaîne vide ''
      copy ( s, -1, 4 )      correct! Donne la chaîne AB
      copy ( s, 3, 4 )       correct! Donne la chaîne CD

      omit ( s, 2, 2 )       donne la chaîne AD
      omit ( s, 2, 3 )       donne la chaîne A
      omit ( s, 3, 2 )       donne la chaîne AB
      omit ( s, -1, 4 )      correct! Donne la chaîne CD
      omit ( s, 2, 4 )       correct! Donne la chaîne A

      include ( sub, s, 3 )  donne la chaîne AB123CD
      include ( sub, s, -2 ) correct! Même effet que concat ( sub, s )
      include ( sub, s, 5 )  correct! Même effet que concat ( s, sub )
```

**Remarques:**

1. Des caractères peuvent être utilisés dans ces fonctions!

2. Dans tous les cas une chaîne **ne peut pas dépasser** 255 caractères.

[ Retour à la table des matières ]

8.2.5.2 **Procédures prédéfinies**

Soient *s, sub* des chaînes de caractères et *indice, nb* des nombres entiers. On a

```
delete ( s, indice, nb )    enlève nb caractères dans s à partir du caractère
                            s [ indice ]

insert ( sub, s, indice )   insère la chaîne sub dans s à partir du caractère
                            s [ indice ]
```

**Exemples** (cf. 8.2.5.1 pour les déclarations et initialisations):

```
delete ( s, 1, 2 );    s vaut alors la chaîne CD
delete ( s, 2, 1 );    s vaut alors la chaîne ACD
delete ( s, -1, 4 );   correct! s vaut alors la chaîne CD
delete ( s, 3, 4 );    correct! s vaut alors la chaîne AB

insert ( sub, s, 3 );  s vaut alors la chaîne AB123CD
insert ( sub, s, -1 ); correct! s vaut alors la chaîne 123ABCD
insert ( sub, s, 5 );  correct!s vaut alors la chaîne ABCD123
```

**Remarques:**

1. Des caractères peuvent être utilisés dans ces procédures!

2. Dans tous les cas la longueur d'une chaîne ne peut pas en dépasser la taille.

[ Retour à la table des matières ]

### 8.2.6 **Entrées-sorties**

Pascal Macintosh permet la lecture et l'écriture de chaînes de caractères au moyen des procédures prédéfinies *read* et *write*. Voici leur sémantique:

```
read ( s );  lit tous les caractères de la ligne depuis la position courante
             jusqu'au caractère de fin de ligne non compris puis affecte cette
             chaîne à la chaîne s (attention à la règle des longueurs, cf. 8.2.4).
```

On peut donc lire une ligne complètement (y compris la marque de fin de ligne) par readln ( *s* ); .

```
write ( s );        écrit la chaîne s sur length ( s ) positions.

write ( s : nb );   écrit la chaîne s sur nb positions. Mais
                    - si nb > length ( s ), la chaîne s est cadrée à droite
                    - si nb < length ( s ), seuls les nb premiers caractères sont
                      écrits
```

**Remarque:**

Les procédures *readln* et *writeln* sont naturellement utilisables avec les chaînes de caractères.

[ Retour à la table des matières ]

### 8.2.7 **Passage en paramètre et résultat de fonction**

Le passage en paramètre de chaînes de caractères peut s'effectuer comme habituellement. Il existe cependant une autre possibilité bien pratique, celle qui est exposée maintenant.

Il est possible d'écrire

```
procedure exemple (     s1 : string;
                    var s2 : string );
...
```

Alors

- le paramètre *s1* est toujours de taille maximale 255 et héritera la longueur du paramètre effectif.
- le paramètre *s2* héritera la longueur et la taille du paramètre effectif.

**Exemple:**

```
const  max = 10;
type   t_chaine = string [ max ];
var    s : t_chaine;
...
  s := '1234';
  exemple ( 'bac', s );
```

Le paramètre *s1* est alors de longueur 3 et vaut *'bac'* à l'appel de *exemple*.
Le paramètre *s2* est alors de taille *max*, de longueur 4 et vaut *'1234'* à l'appel de *exemple*.

Le type du résultat d'une fonction peut également être mentionné comme **string**:

```
function exemple : string;
...
```

**Exemple**:

```
function exemple : string;
begin (* exemple *)
   exemple := 'abc';
end; (* exemple *)
...
s := exemple;         (* s vaut alors 'abc', est donc de longueur 3 *)
```

[ Retour à la table des matières ]

**ANNEXE G**

# DIAGRAMMES SYNTAXIQUES DE PASCAL

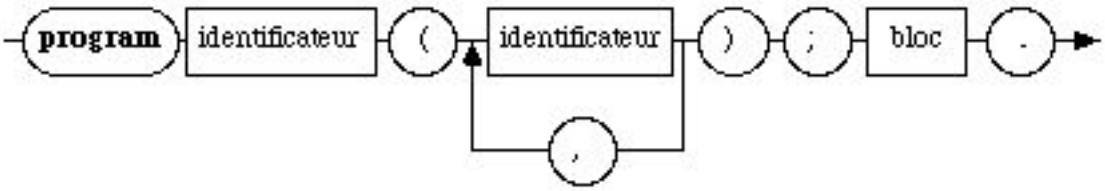
---



[ Retour à la table des matières ]

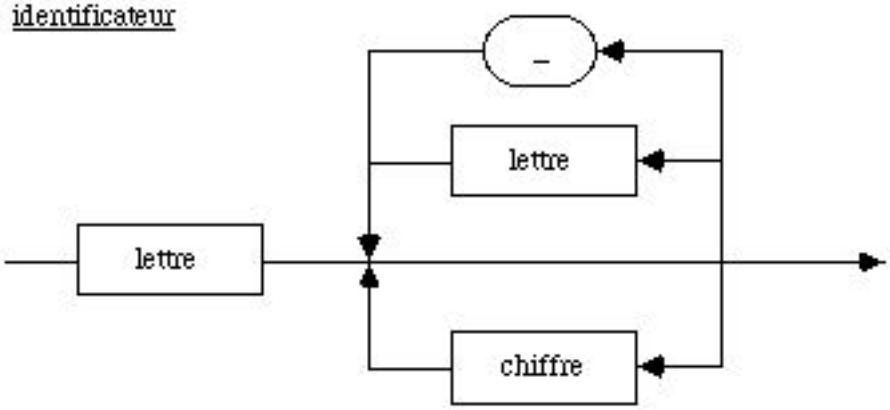
---

programme



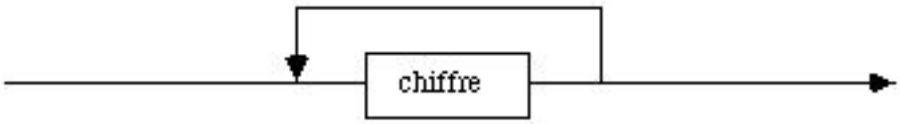

[ Retour au sommaire ]

identificateur



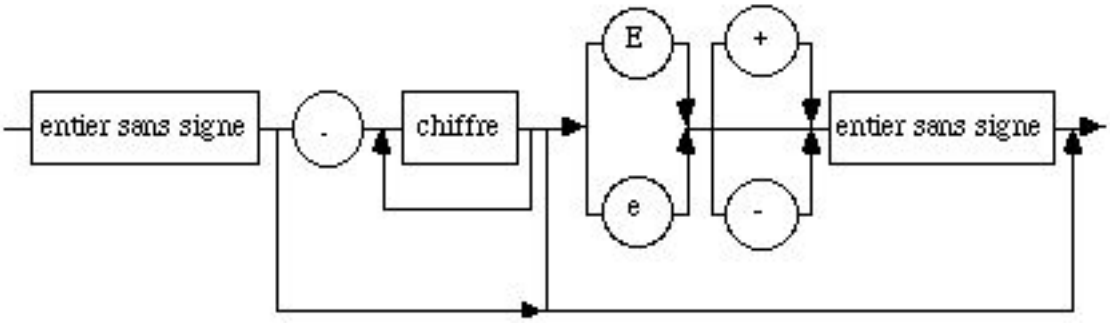

[ Retour au sommaire ]

entier sans signe



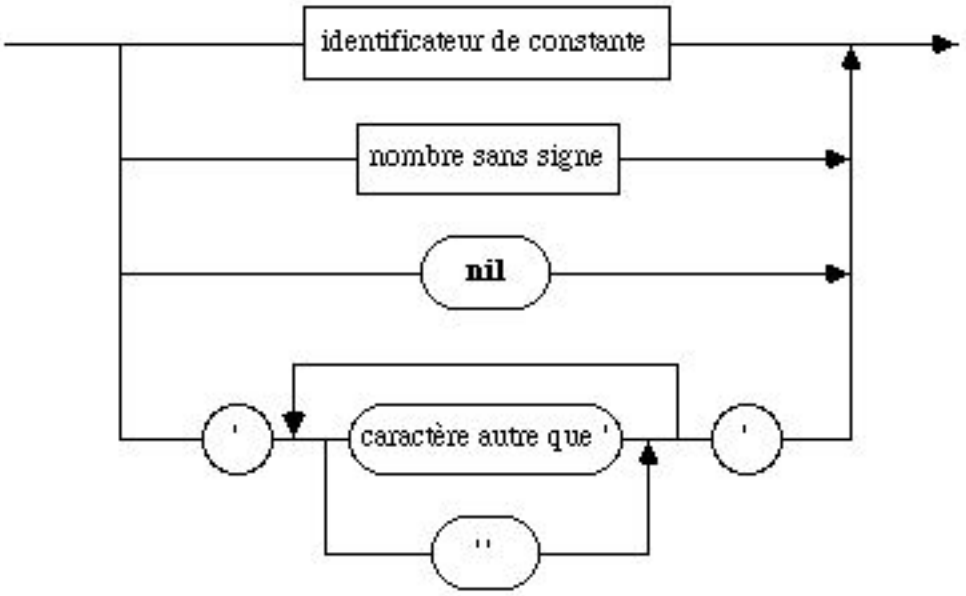

[ Retour au sommaire ]

nombre sans signe



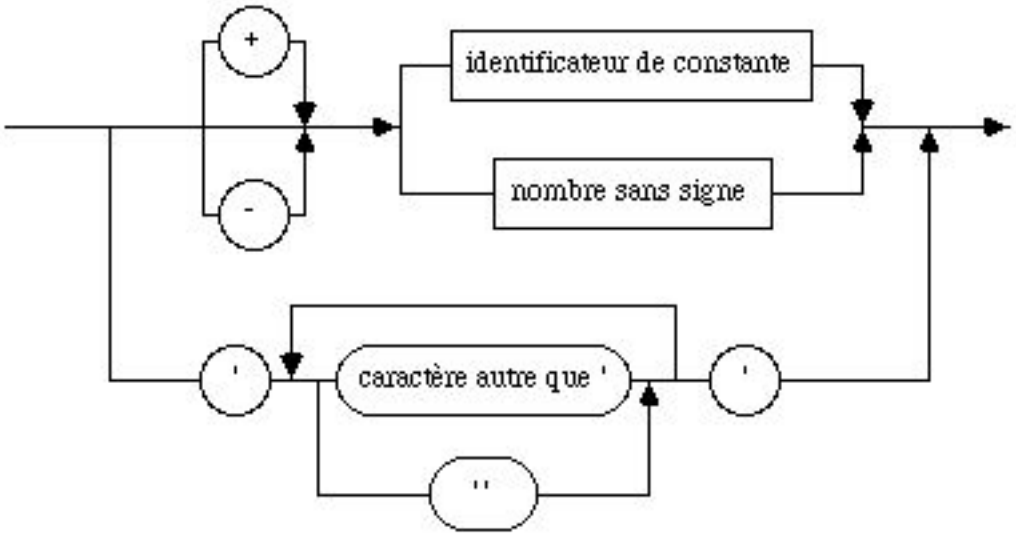

[ Retour au sommaire ]

constante sans signe



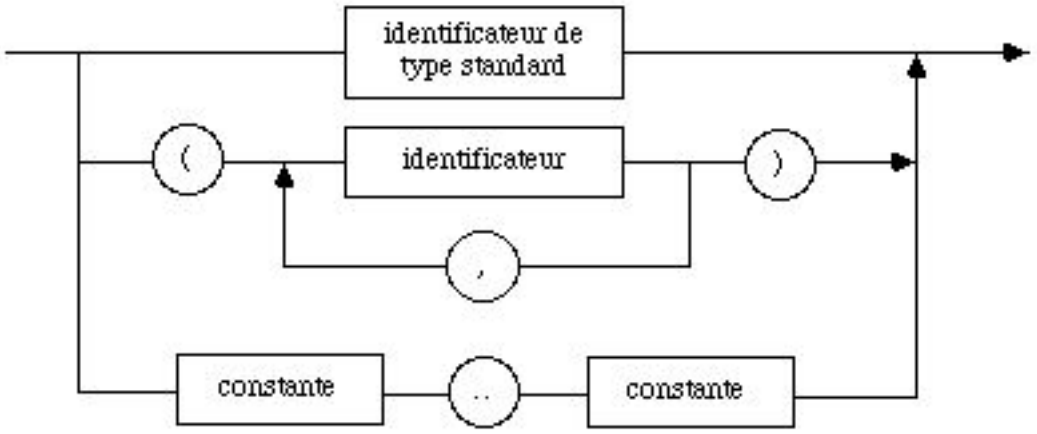

[ Retour au sommaire ]

constante



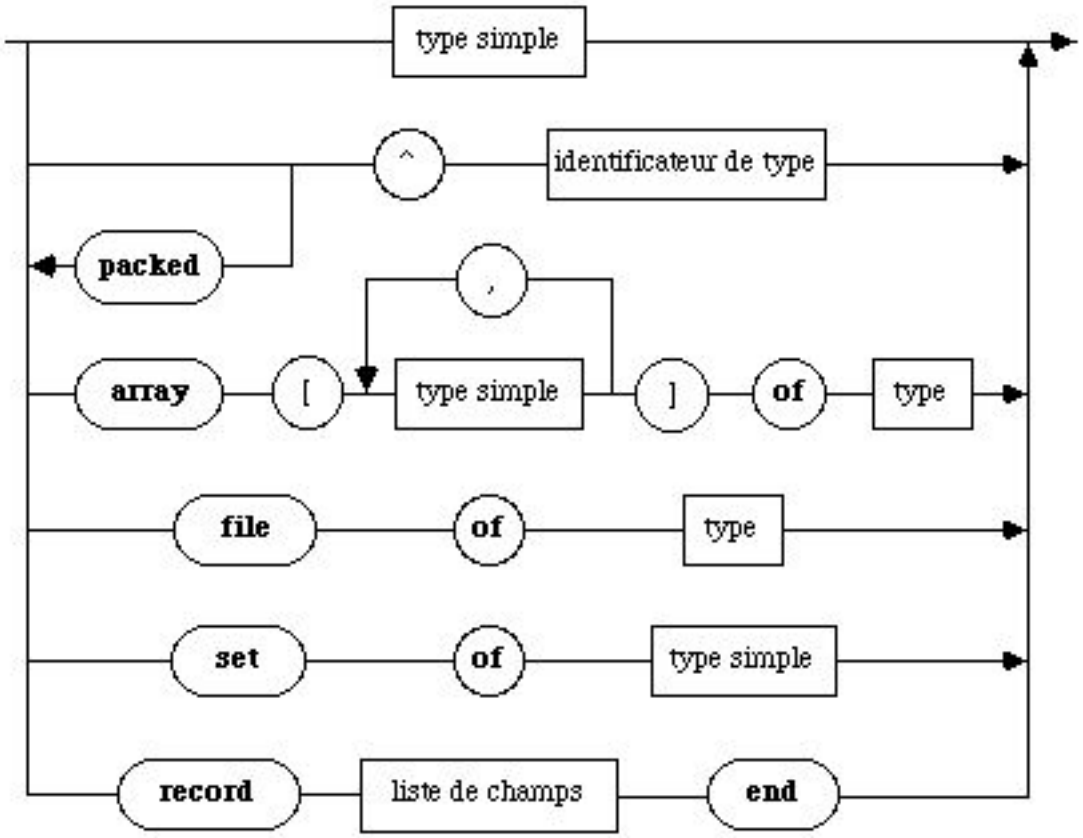

[ Retour au sommaire ]

type simple



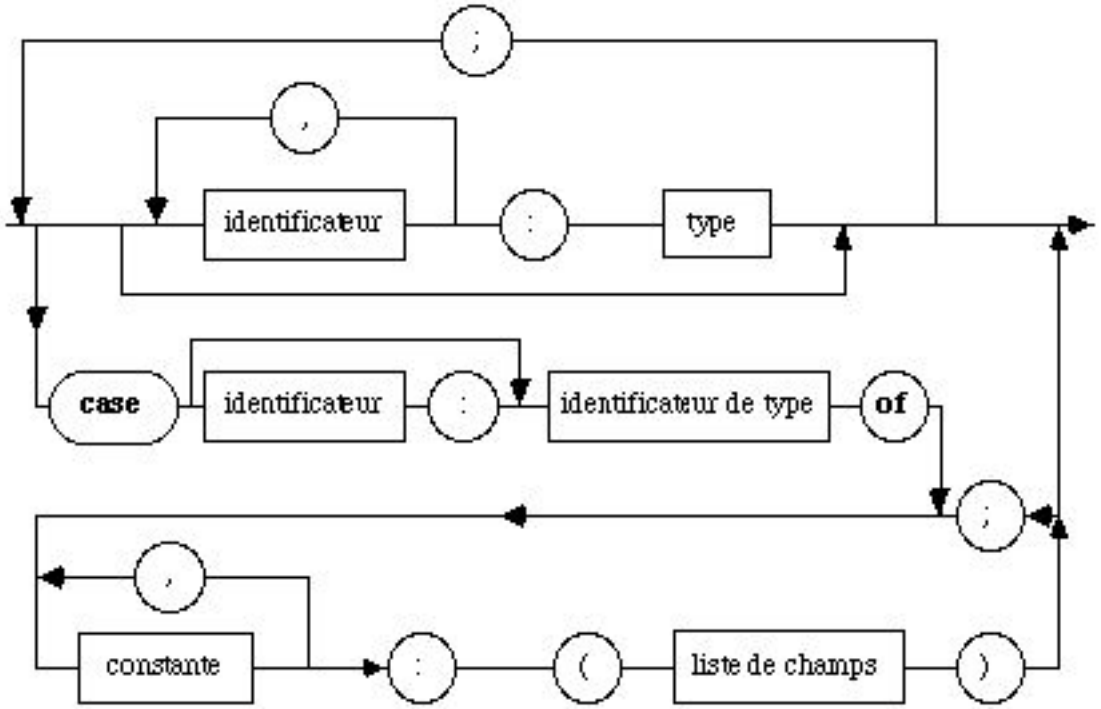

[ Retour au sommaire ]

type



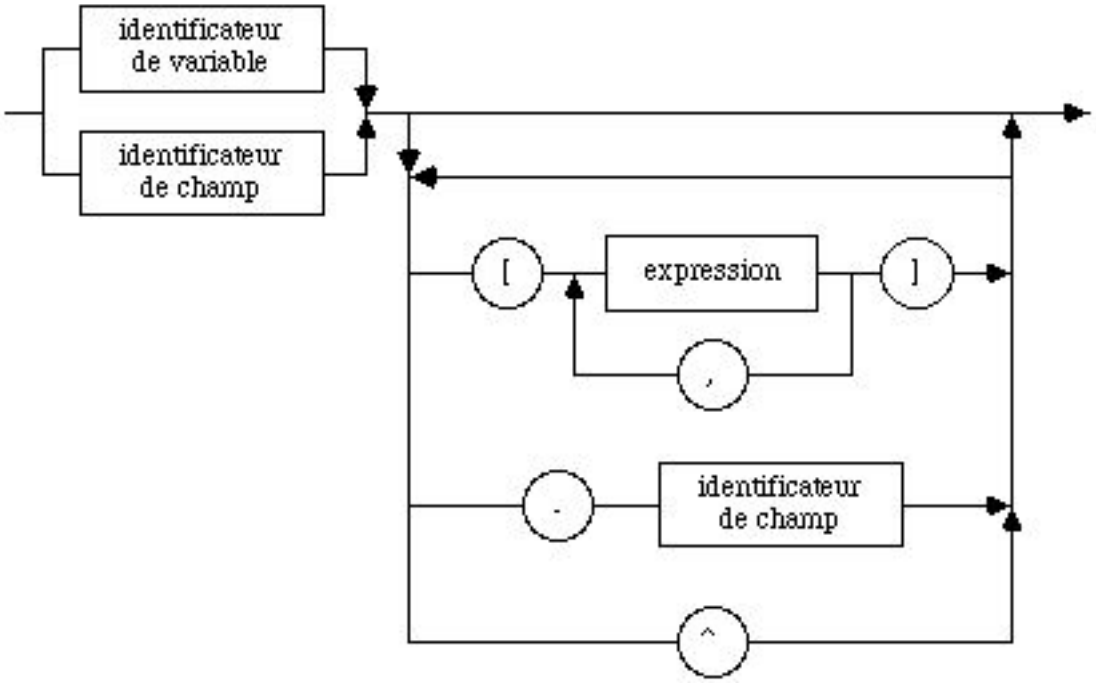

[ Retour au sommaire ]

liste de champs



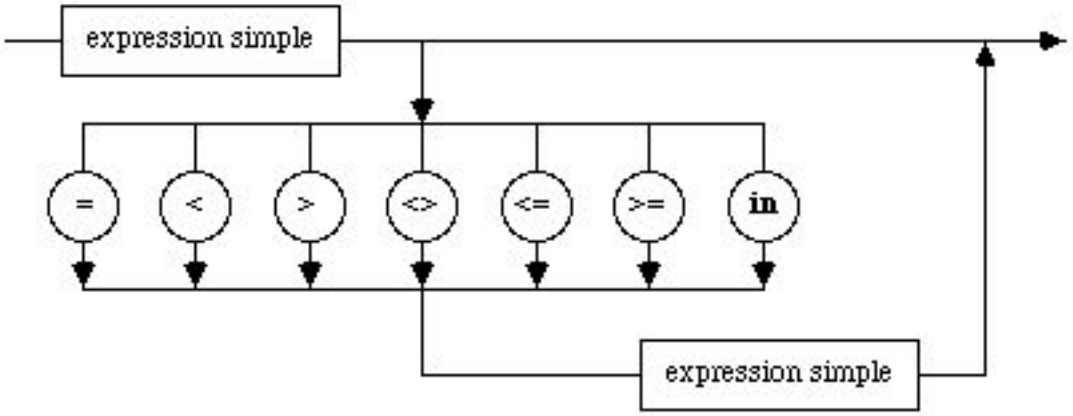

[ Retour au sommaire ]

variable



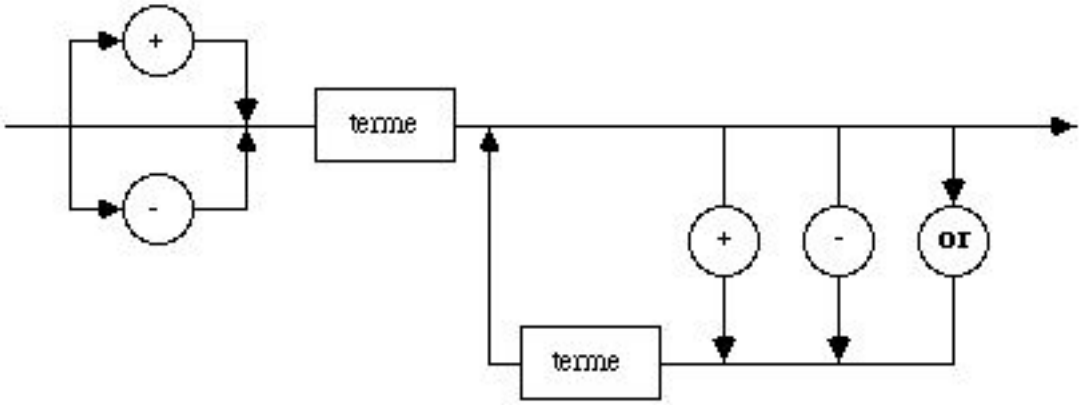

[ Retour au sommaire ]

expression



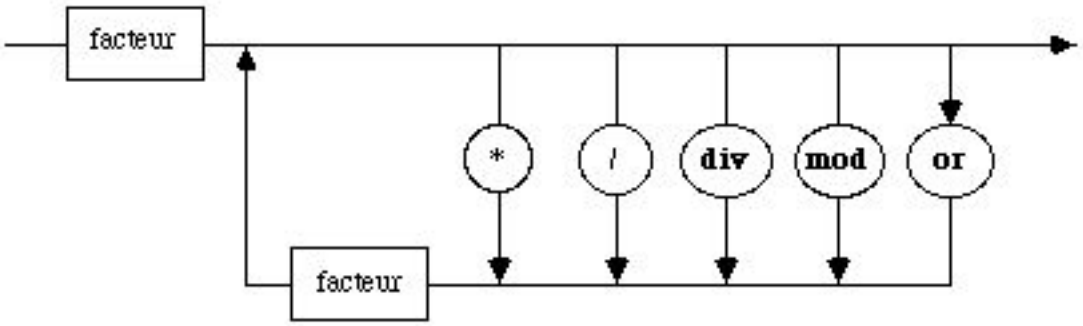

[ Retour au sommaire ]

expression simple



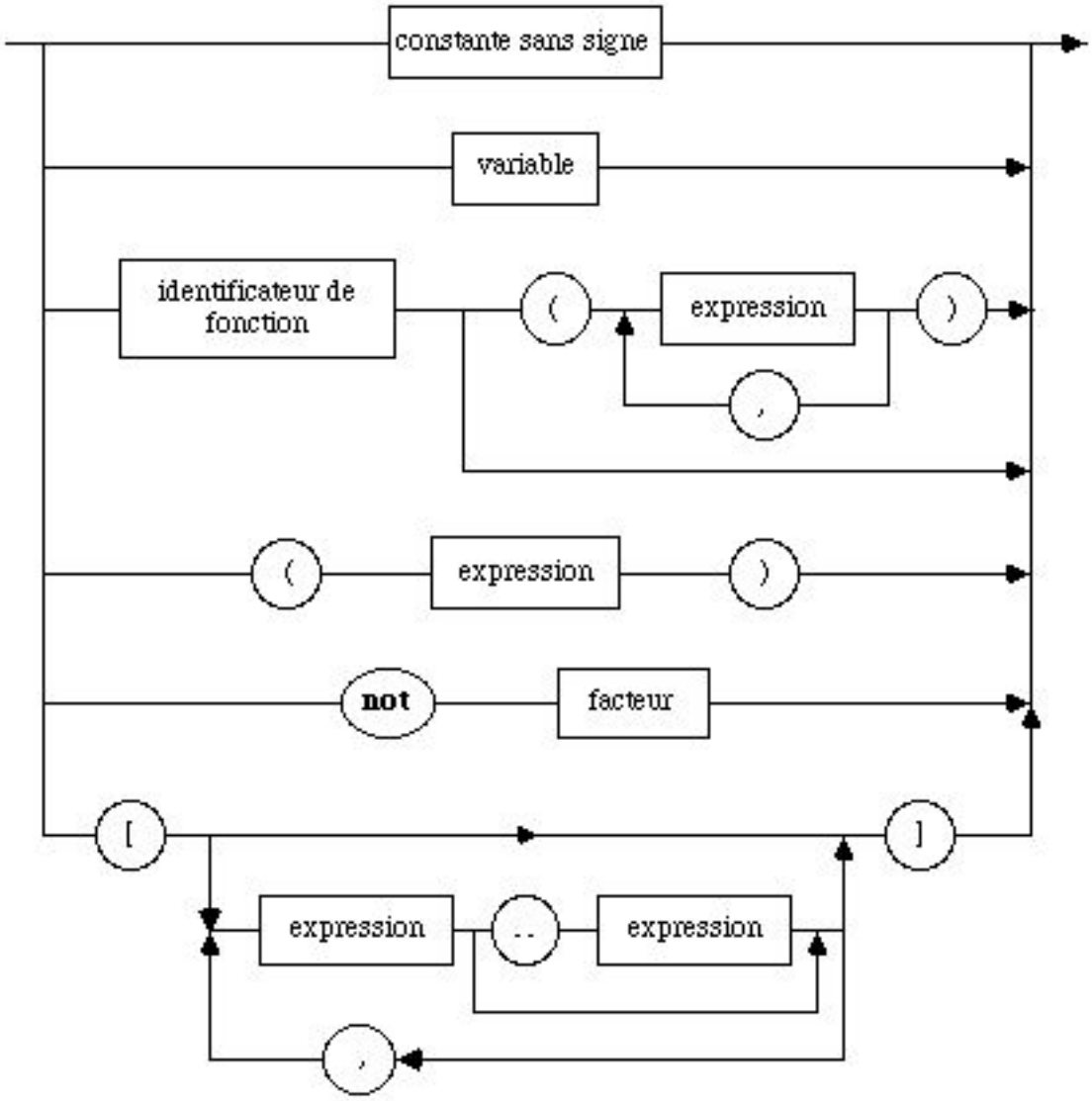

[ Retour au sommaire ]

terme



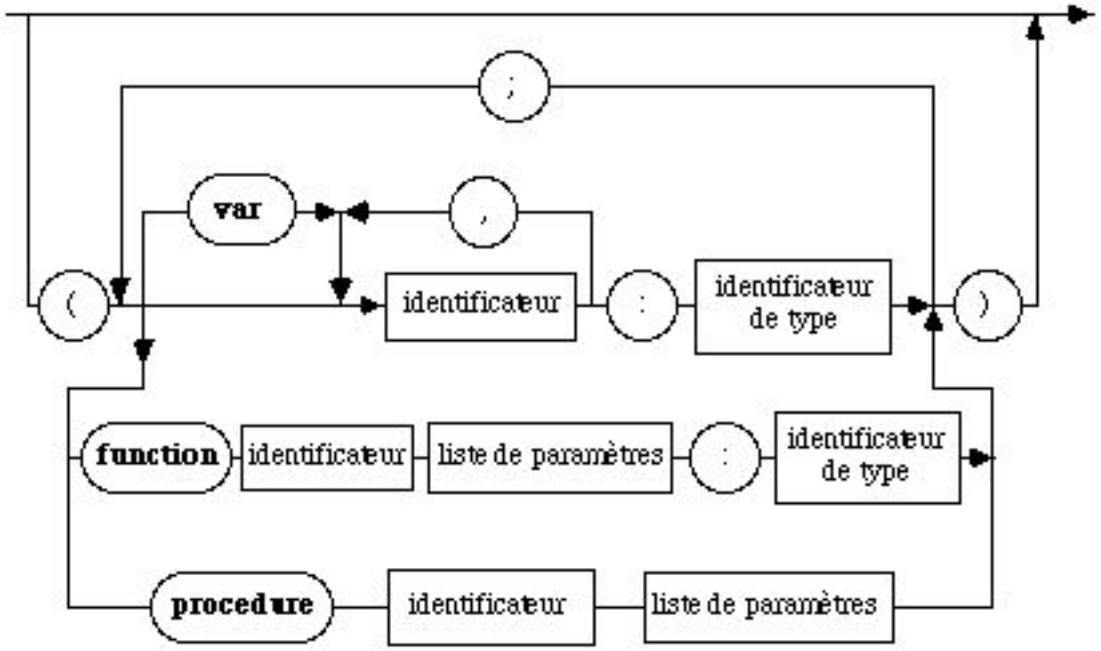

[ Retour au sommaire ]

facteur



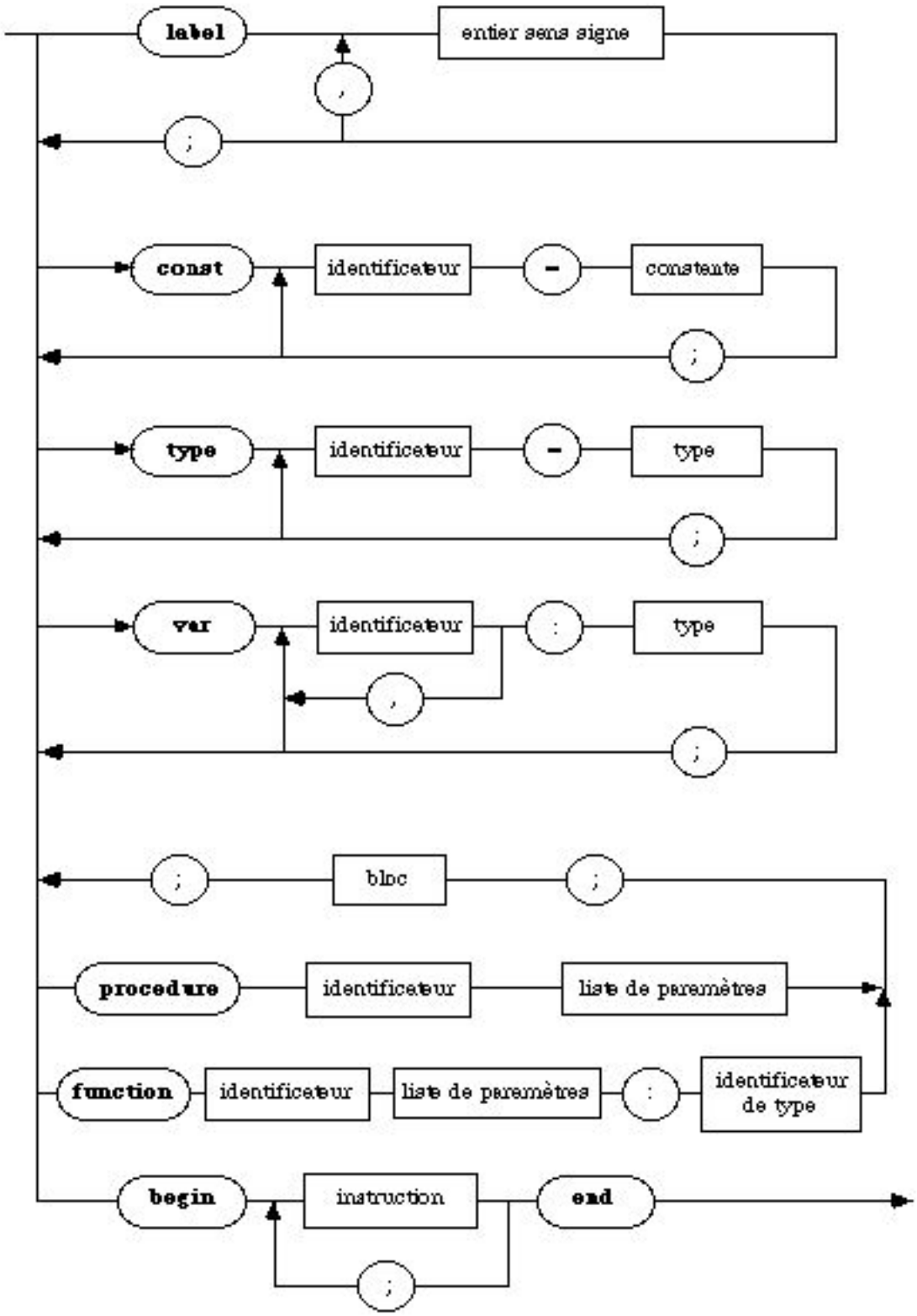

[ Retour au sommaire ]

liste de paramètres

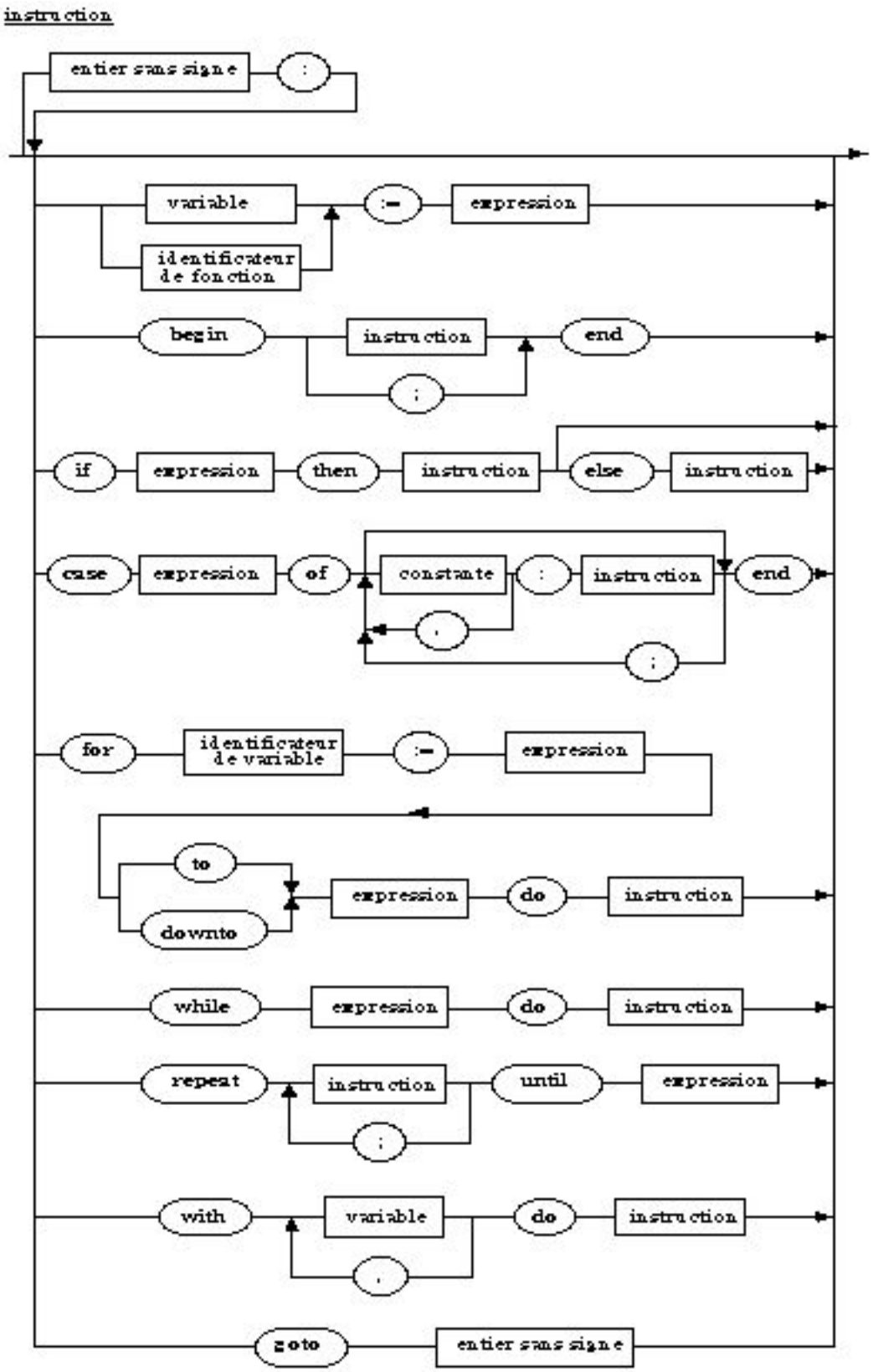



[ Retour au sommaire ]

bloc



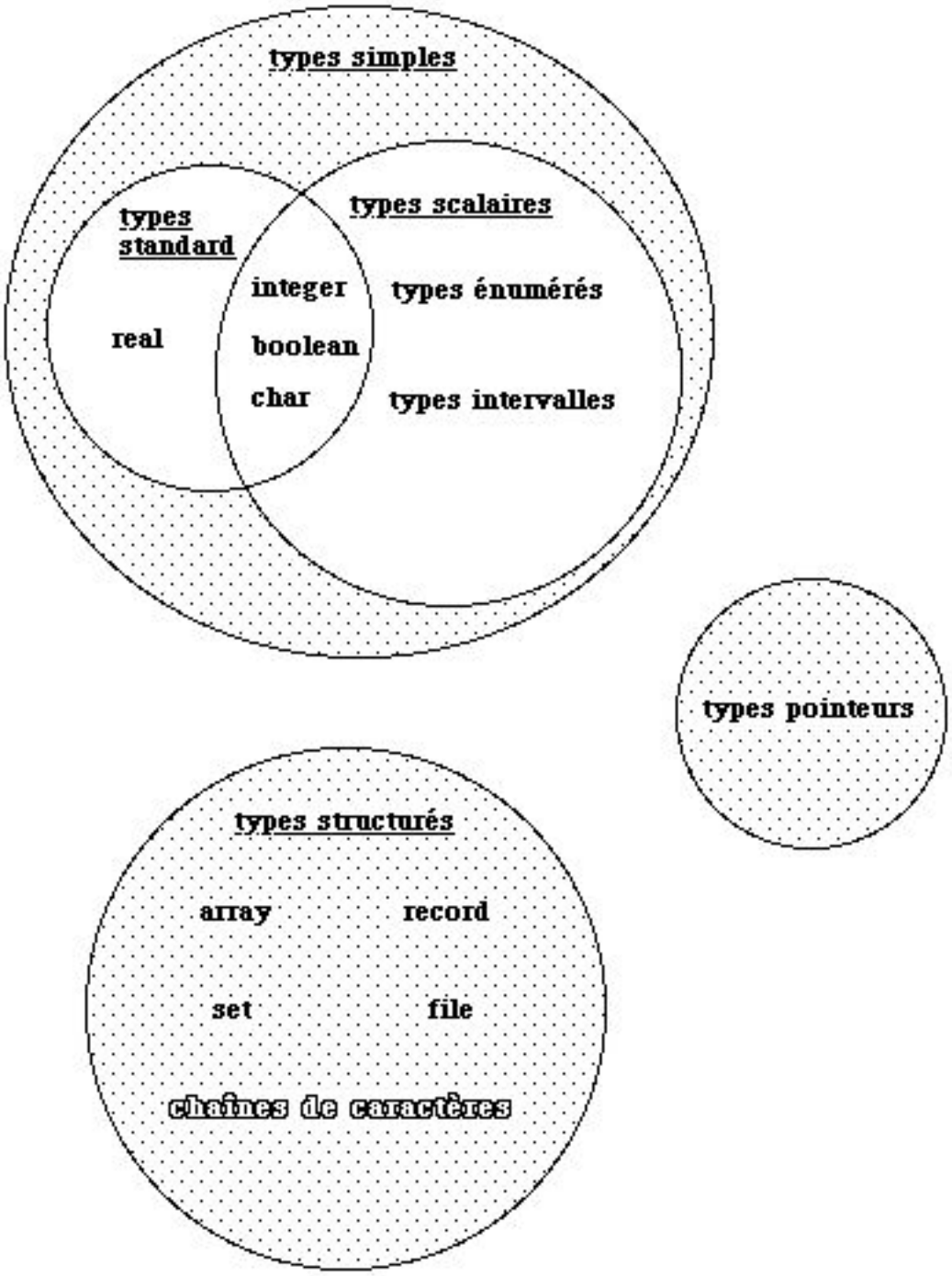

[ Retour au sommaire ]




[ Retour au sommaire ]

**ANNEXE F**

# CLASSIFICATION DES TYPES EN PASCAL




**Remarque :**

Chaque version de Pascal possède son type **chaînes de caractères**!

[ Retour à la table des matières ]

**ANNEXE E**

# PROCEDURES ET FONCTIONS PREDEFINIES EN PASCAL

## PROCEDURES PREDEFINIES

| Appel de la procedure | Actions effectuées |
|---|---|
| dispose (pt); | rendre au tas la zone mémoire pointée par *pt^*. Après exécution, *pt* a une valeur indéfinie |
| dispose ( pt, c1, c2, ..., cn ); | doit être utilisée à la place de *dispose ( pt )* si *pt* a été créée par *new ( pt, c1, c2, ..., cn )* (même *ci* !) |
| get ( f ); | lit l'élément suivant du fichier *f* ouvert en lecture ( possible que si *eof ( f )* est fausse !) et l'affecte à *f^* |
| new ( pt ); | alloue une zone mémoire, pointée par *pt* |
| new ( pt, c1, c2, ..., cn ); | alloue une zone mémoire, pointée par *pt* . Cette forme doit être utilisée dans le cas des enregistrements à variante, pour ne réserver que la place strictement nécessaire aux variantes correspondant aux valeurs *ci* |
| pack ( t1, i, t2 ); | soit t1 de type **array** [m1..n1] **of** T<br>et t2 de type **packed array** [m2..n2] **of** T<br>alors l'effet de *pack* est équivalent à l'instruction<br>**for** j:= m2 **to** n2 **do** t2[j] := t1 [j - m2 + i]; |
| page ( f ); | provoque un saut de page sur le fichier (de caractères) *f* |
| put ( f ); | écrit l'élément *f^* à la fin du fichier *f* ouvert en écriture |
| read (f, v1, ..., vn ); | lit n valeurs dans le fichier de caractères *f* et les affecte à *v1, v2..., vn.* Si *f* est omis, cela équivaut à *read ( input, v1, ..., vn );* |
| readln ( f, v1, ..., vn ); | comme *read (...);* mais passe ensuite à la ligne suivante |
| reset ( f ); | prépare la lecture du fichier *f* à partir du début de *f* |
| rewrite ( f ); | réinitialise le fichier *f* et le prépare pour l'écriture |

| | |
|---|---|
| unpack ( t2, t1, i ); | soit t1 de type **array** [m1..n1] **of** T<br>et t2 de type **packed array** [m2..n2] **of** T<br>alors l'effet de *unpack* est équivalent à l'instruction<br>**for** j:= m2 **to** n2 **do** t1[j - m2 + i] := t2[j]; |
| write ( f, v1, ..., vn ); | écrit les n valeurs *v1, v2, ..., vn* dans le fichier *f* . Si *f* est omis, cela équivaut à *write ( output, v1, ..., vn );* |
| writeln ( f, v1, ..., vn ); | comme *write (...);* mais passe ensuite à la ligne suivante |

**FONCTIONS PREDEFINIES**

| **Appel de fonction** | **Actions effectuées** |
|---|---|
| abs ( valeur_numerique )<br>de | donne la valeur absolue ( de type *integer,* resp. *real* )<br>*valeur_numerique* ( de type *integer,* resp. *real* ) |
| arctan ( angle_en_radians )<br>type | arctg(*angle_en_radians* ) *(angle _en_radians* de *real)* avec - pi/2 <= *angle_en_radians* <= pi/2 |
| chr ( valeur_entiere ) | donne le caractère dont le code ASCII est *valeur _entiere* ( entier compris entre 0 et 127 ( ou 255 )) |
| cos ( angle_en_radians ) | cosinus de *angle_en_radians* ( de type *real* ) |
| eof ( f ) | donne la valeur *true* si la fin du fichier *f* est atteinte, *false* sinon. L'abréviation *eof* signifie *eof ( input )* |
| eoln ( f ) | donne la valeur *true* si la fin de la ligne courante du fichier *f* est atteinte, *false* sinon. L'abréviation *eoln* signifie *eoln ( input )* |
| exp ( x ) | "e puissance x" avec x de type *integer* ou *real* |
| ln ( x ) | "log naturel de x" avec x de type *integer* ou *real* |
| odd ( valeur ) | donne la valeur *true* si *valeur* (de type *integer* ) est impaire, *false* sinon |
| ord ( valeur ) | - donne le code ASCII de *valeur* si *valeur* de type *char*<br>- donne le numéro d'ordre de *valeur* si *valeur* est d'un type énuméré<br>- donne 0 si *valeur* vaut *false,* 1 si la *valeur* vaut *true*<br>- donne l'adresse de l'élément pointé par *valeur* si *valeur* est d'un type pointeur |
| pred ( valeur ) | donne le prédécesseur de *valeur* ( d'un type scalaire ). Provoque une erreur si le prédécesseur n'existe pas! |

| | |
|---|---|
| round ( valeur ) | donne l'arrondi (de type *integer* ) de *valeur* (de type *real* ) |
| sin ( angle_en_radians ) | sinus de *angle_en_radians* (de type *real* ) |
| sqr ( valeur ) | donne le carré (de type *integer* , resp. *real* ) de *valeur* (de type *integer*, resp. *real* ) |
| sqrt ( valeur ) | donne la racine carrée (de type *real* ) de *valeur* (de type *integer* ou *real* ). Valeur doit être >=0! |
| succ ( valeur ) | donne le successeur de *valeur* (d'un type scalaire). Provoque une erreur si le successeur n'existe pas! |
| trunc ( valeur ) | donne la partie entière (de type *integer*, ) de *valeur* (de type *real* ) |

**Remarque :**

Chaque version de Pascal comporte des procédures et fonctions prédéfinies supplémentaires !

[ Retour à la table des matières ]

**CHAPITRE 7**

# LES TYPES ENUMERES ET INTERVALLE

## 7.1 Les types énumérés

### 7.1.1 Motivation

Les types énumérés permettent de représenter des valeurs en les énumérant au moyen de leur nom. Supposons que nous voulons représenter les jours de la semaine. Jusqu'à présent, la seule façon de faire était de déclarer sept constantes de la manière suivante:

```
const  lundi = 1;
       mardi = 2;
       ...
```

Ceci peut conduire à des erreurs, être fastidieux à déclarer. Laissons cela aux malheureux programmeurs Fortran et voyons ce que proposent des langages plus récents.

[ Retour à la table des matières ]

### 7.1.2 Généralités

Dans les langages modernes les déclarations de types énumérés permettent de construire **explicitement** les valeurs de ces types. Nous pouvons écrire:

```
type t_jours_de_la_semaine = ( lundi, mardi, mercredi, jeudi, vendredi, samedi,
                               dimanche );
```

Ceci signifie que nous déclarons et utiliserons le type *jours_de_la_semaine* formé des valeurs *lundi*, *mardi* ... *dimanche*. Deux autres exemples traditionnels:

```
type t_couleurs_arc_en_ciel = ( rouge, orange, jaune, vert, bleu, indigo, violet
);

       t_mois_de_l_annee = ( janvier, fevrier, mars, avril, mai, juin, juillet,
aout,
                             septembre, octobre, novembre, decembre );
```

Les **constantes** d'un type énuméré sont par définition les **identificateurs** énumérés entre les parenthèses.

Les **opérations** possibles (en plus de l'affectation et du passage en paramètre) sur les valeurs d'un type énuméré sont:

```
=   <>   <   <=   >   >=
```

Les valeurs d'un type énuméré sont en effet **ordonnées** selon leur ordre de déclaration. Pour le type *t_jours_de_la_semaine* nous avons

lundi < mardi < mercredi < jeudi < vendredi < samedi < dimanche

Les **expressions** se limitent aux constantes, variables et fonctions à résultat du type énuméré considéré.

**Remarques:**

1. A chaque valeur énumérée correspond un **numéro d'ordre** (nombre entier). La première valeur porte le numéro 0, la seconde le numéro 1 etc. Dans notre exemple

lundi porte le numéro 0
mardi porte le numéro 1 ...

2. Les types énumérés font partie des types scalaires (cf. annexe F).

[ Retour à la table des matières ]

### 7.1.3 **Affectation**

L'affectation se fait de manière habituelle:

```
var demain,
     jour : t_jours_de_la_semaine; (* cf. 7.1.2 *)
...
  jour := lundi;
  demain := succ ( jour );      (* demain vaut alors mardi, cf. 7.1.4 *)
```

[ Retour à la table des matières ]

### 7.1.4 **Fonctions prédéfinies**

Les fonctions prédéfinies sur les types énumérés sont:

| | |
|---|---|
| `succ ( expr_type_enumere )` | fournit le **successeur** de la valeur donnée par *expr_type_enumere* (erreur si cette expression désigne la dernière valeur énumérée) |
| `pred ( expr_type_ enumere )` | fournit le **prédécesseur** de la valeur donnée par *expr_type_enumere* (erreur si cette expression désigne la première valeur énumérée) |
| `ord ( expr_type_ enumere )` | fournit le **numéro d'ordre** de la valeur donnée par *expr_type_enumere* |

[ Retour à la table des matières ]

### 7.1.5 **Entrées-sorties**

Le Pascal standard n'admet aucune entrée-sortie sur des valeurs d'un type énuméré. Cependant plusieurs compilateurs fournissent cette extension, comme par exemple Pascal Macintosh et Pascal VAX. Dans ces deux cas la sémantique est la suivante:

- la lecture et l'écriture d'une valeur d'un type énuméré s'effectuent comme la lecture et l'écriture d'un nombre entier.

**Exemples:**

```
var jour : t_jours_de_la_semaine; (* pour notre exemple *)
...
  read ( jour );
  write ( lundi );       (* écrit la constante lundi sur le nombre minimum *)
                         (* de positions, à savoir 5 *)
  write ( jour : 10 );   (* si jour a la valeur mercredi on obtient l'écriture *)
                         (* de deux espaces suivis du mot mercredi *)
```

[ Retour à la table des matières ]

### 7.1.6 **Types énumérés, tableaux et instruction for**

Un type énuméré peut être utilisé pour définir les indices d'un tableau (cf. 6.3.2).

Par exemple:

**type** t_heures_a_travailler = **array** [ t_jours_de_la_semaine ] **of** real;

Le type *t_heures_a_travailler* représente un tableau de 7 éléments numérotés *lundi, mardi, ..., dimanche* (cf. 7.1.2).

Un type énuméré peut aussi être utilisé dans une boucle **for** (cf. 4.1.2).

Par exemple:

```
for jour := lundi to vendredi do ...;   (* voir déclarations ci-dessus *)
```

[ Retour à la table des matières ]

## 7.2 **L'instruction case**

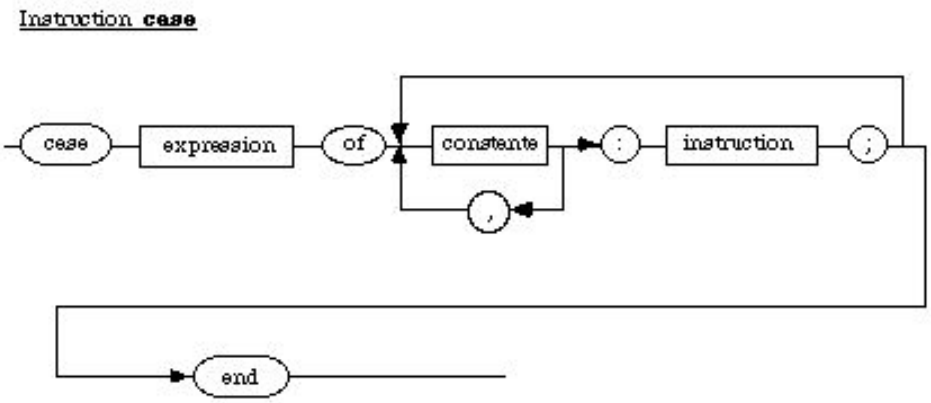
Le paragraphe traitant l'instruction **if** (cf. 4.2) nous a montré comment programmer un choix, limité à deux branches. Nous allons généraliser cela en montrant comment programmer un **choix multiple**, réalisé par l'instruction **case**.

Supposons que nous voulions poser une question à l'utilisateur du programme. Celui-ci nous répond par un chiffre et l'exécution du programme se poursuit en fonction de ce chiffre. Jusqu'à présent ce problème serait résolu par une succession d'instructions **if** imbriquées. La façon suivante de procéder est meilleure:

```
var reponse : integer; (* nombre donné par l'utilisateur *)
...
  case   reponse of
     1   : begin
             (* traitement si l'utilisateur répond 1 *)
           end;
     2   : begin
             (* traitement si l'utilisateur répond 2 *)
           end;
 3,4,5   : begin
             (* traitement si l'utilisateur répond 3, 4 ou 5 *)
           end;
     6   : begin
```

```
            (* traitement si l'utilisateur répond 6 *)
          end;
 7,8,9    : begin
            (* traitement si l'utilisateur répond 7, 8 ou 9 *)
          end;
end; (* case *)
```

La forme générale de l'instruction case est donnée par le diagramme syntaxique suivant:




expression doit être d'un type scalaire (cf. annexe F). Il n'est donc **pas** possible d'écrire par exemple

```
case chaine of (* avec chaine d'un type chaîne de caractères *)
   'oui' : ...; (* cf 8.2.2 *)
   'non' : ...;
end; (*case *)
```

**Remarques:**

1. Une constante représentant une valeur possible de l'expression ne peut apparaître qu'**une seule fois.**

2. Si l'expression a une valeur ne correspondant à aucune des constantes mentionnées, le comportement du programme dépend du compilateur!
   Avec Pascal Macintosh et Pascal VAX une erreur est signalée et l'exécution est interrompue.

3. Pour résoudre ce problème certains compilateurs Pascal (comme Pascal Macintosh et Pascal VAX) offrent la possibilité suivante:

```
case   reponse of
   1   : begin
           (* traitement si l'utilisateur répond 1 *)
         end;
   2   : begin
           (* traitement si l'utilisateur répond 2 *)
         end;
3,4,5  : begin
           (* traitement si l'utilisateur répond 3, 4 ou 5 *)
         end;
   6   : begin
           (* traitement si l'utilisateur répond 6 *)
         end;
7,8,9  : begin
           (* traitement si l'utilisateur répond 7, 8 ou 9 *)
```

```
          end;
  otherwise begin
              (* traitement si reponse a une valeur différente de 1, 2,..., 9 *)
              (* ATTENTION: pas de symbole : après otherwise *)
            end;
  end; (* case *)
```

4. L'intruction composée **begin** ... **end** n'est bien sûr pas nécessaire lorsqu'une branche ne comporte qu'une seule instruction.

[ Retour à la table des matières ]

## 7.3 **Les types intervalle**

### 7.3.1 **Généralités**

Un **intervalle** permet de **restreindre le groupe des valeurs** d'un type appelé **type de base** et choisi parmi *integer, boolean, char* ou un type énuméré.

**Exemples:**

- 0..9 est l'intervalle des valeurs entières 0 à 9, le type de base est *integer*
- lundi..vendredi est l'intervalle des valeurs énumérées *lundi* à *vendredi*, le type de base est *jours_de_la_semaine*
- 'A'..'Z' est l'intervalle des lettres majuscules, le type de base est *char*

Les bornes inférieure et supérieure appartiennent à l'intervalle ainsi défini. Ce sont obligatoirement des **constantes du type de base**.

La déclaration d'un type intervalle se fait ainsi:

```
type   t_chiffre = 0..9;
       t_jours_de_travail = lundi..vendredi;
       t_majuscule = 'A'..'Z';
```

Les **constantes** d'un type intervalle sont celles du type de base comprises dans l'intervalle (bornes incluses).

Les **opérations** possibles sont celles du type de base.

Les **expressions** se construisent comme celles du type de base.

La forme générale de la déclaration d'un type intervalle est donc

***type identificateur_de_type_intervalle = intervalle;***

où ***intervalle* est de la forme *constante .. constante***

**Remarque:**

Les types intervalle font partie des types scalaires (cf. annexe F).

[ Retour à la table des matières ]

### 7.3.2 **Affectation**

D'une part l'affectation se fait de manière habituelle:

```
var jour : t_jours_de_travail; (* cf. 7.3.1 *)
...
  jour := lundi;
```

D'autre part un type intervalle et son type de base sont **compatibles**, c'est-à-dire que:

- l'affectation d'une valeur de type intervalle à une variable du type de base est toujours possible

- l'affectation d'une valeur du type de base à une variable du type intervalle est possible si cette valeur appartient à l'intervalle (une erreur est signalée à l'exécution dans le cas contraire)

- la construction d'expressions mixtes est possible, la valeur d'une telle expression appartenant alors au type de base

**Exemples:**

```
- var majuscule : t_majuscule; (* cf. 7.3.1 *)
      lettre : char;
  ...
      lettre := majuscule; (* toujours correct *)
      majuscule := lettre; (* correct si lettre contient une majuscule *)

- var prix_unitaire : t_chiffre; (* cf. 7.3.1 *)
      quantite : integer;
  ...
      prix_unitaire := 4;
      quantite := 100;
      writeln ( 'Le prix est de ', quantite * prix_unitaire, ' francs' );
      (* l'expression quantite * prix_unitaire est de type integer *)
```

[ Retour à la table des matières ]

### 7.3.3 **Fonctions prédéfinies et entrées-sorties**

Les fonctions prédéfinies ainsi que les entrées-sorties sont celles définies sur les valeurs du type de base.

[ Retour à la table des matières ]

### 7.3.4 **Types intervalle, tableaux et instruction for**

Un type intervalle peut être utilisé pour définir les indices d'un tableau (cf. 6.3.5).

Par exemple:

```
const max_elements = 10;

type  t_indice_table = 1..max_elements;
      t_table = array [ t_indice_table ] of integer;
```

ce qui est d'ailleurs la meilleure déclaration d'un type tableau!

Un type intervalle peut aussi être utilisé dans une boucle **for** (cf. 4.1.2).

Par exemple:

```
   var indice : t_indice_table;     (* voir déclarations ci-dessus *)

   ...
      for indice := 1 to max_elements do ...;
```

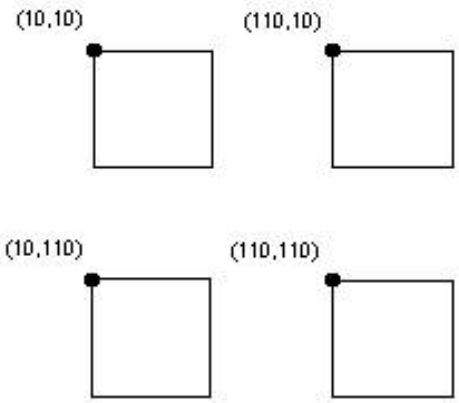
[ Retour à la table des matières ]

**CHAPITRE 5**

# LES PROCEDURES ET LES FONCTIONS

## 5.1 **Motivation**

Supposons que nous voulons effectuer le dessin suivant (sans les coordonnées des points, mises pour fixer les idées):




La **décomposition** (cf. 2.2) de ce problème simple nous ramène à quatre dessins de carrés **identiques**, disposés à quatre endroits différents. Un langage tel que Pascal nous fournit un moyen de ne pas répéter quatre fois les mêmes instructions de dessin d'un carré. Ce moyen est une construction appelée **procédure** (sans paramètre). Si les instructions de dessin sont (Pascal Macintosh):

```
line ( 50, 0 );   (* dessin d'un carré de 50 de côté. L'instruction *)
line ( 0, 50 );   (* line(dx,dy); dessine un trait depuis la position *)
line ( -50, 0 );  (* courante (x,y) jusqu'au point (x+dx,y+dy) qui *)
line ( 0, -50 );  (* devient la position courante. *)
```

la procédure dessinant un tel carré est:

```
procedure dessiner_carre;
(* Cette procédure dessine un carré *)
begin (* dessiner_carre *)
   line ( 50, 0 );  (* dessin d'un carré de 50 de côté. L'instruction   *)
   line ( 0, 50 );  (* line(dx,dy); dessine un trait depuis la position *)
   line ( -50, 0 ); (* courante (x,y) jusqu'au point (x+dx,y+dy) qui    *)
   line ( 0, -50 ); (* devient la position courante. *)
end; (* dessiner_carre *)
```

Une procédure n'est pas exécutée pour elle-même. C'est le programme principal (ou une autre procédure) qui **déclare** la procédure et qui **l'appelle**. L'appel de la procédure *dessiner_carre* remplace les 4 instructions line (...). Cet appel est une instruction qui s'écrit en utilisant le **nom** de la procédure:

```
dessiner_carre;
```

Les lecteurs attentifs et malins auront remarqué que l'instruction *line (...)*; est un appel de la procédure *line* suivi de ce que l'on appelle des paramètres (cf. 5.4). Remarquons à ce propos que l'identificateur *line* ne suit pas la convention sur les noms des procédures (voir document "Présentation des programmes").

Le programme complet réalisant le dessin proposé est donc:

```
program dessin_quatre_carres ( input, output );
(* Auteur:...*)

   (* les procédures se déclarent comme les constantes et *)
   (* les variables mais après celles-ci *)

   procedure dessiner_carre;
   (* Cette procédure dessine un carré *)

      const cote = 50; (* longueur d'un côté du carré *)

   begin (* dessiner_carre *)
      line ( cote,0 );   (* dessin d'un carré de 50 de côté. L'instruction *)
      line ( 0,cote );   (* line(dx,dy); dessine un trait depuis la position *)
      line ( -cote,0 );  (* courante (x,y) jusqu'au point (x+dx,y+dy) qui *)
      line ( 0,-cote );  (* devient la position courante. *)
   end; (* dessiner_carre *)

   begin (* dessin_quatre_carres *)

      showdrawing;        (* ouvrir la fenêtre de dessin *)

      moveto ( 10,10 );   (* déplacement au point (10,10) *)
      dessiner_carre;     (* dessin du carré depuis ce point... *)
      moveto ( 110,10 );  (* ... et de même pour les autres. *)
      dessiner_carre;
      moveto ( 110,110 );
      dessiner_carre;
      moveto ( 10,110 );
      dessiner_carre;

   end.
```

On remarquera la lisibilité de ce programme. Les procédures (et les fonctions) sont des constructions qui

- améliorent la **lisibilité** des programmes, donc aident à la **compréhension** d'un programme;
- reflètent la **décomposition** d'un problème en sous-problèmes plus faciles à analyser et à résoudre;
- diminuent la **taille** des programmes, donc le temps de saisie (!);
- réduisent les **risques d'erreurs.**

Une procédure (ou une fonction) peut être vue comme une **boîte noire**. Le programmeur veut utiliser la boîte noire sans se préoccuper de sa structure interne. Ceci facilite l'écriture des programmes, en particulier ceux de grande taille.

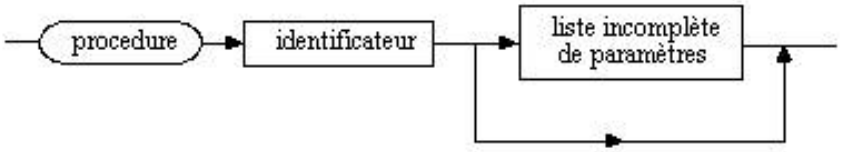
[ Retour à la table des matières ]

## 5.2 **Structure des procédures**

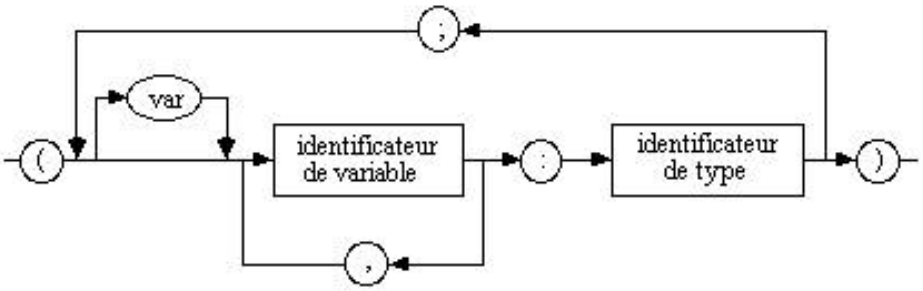
Une procédure a la même structure qu'un programme principal (cf. 2.4), à savoir un **en-tête,** une **partie déclarative** et un

**corps**. Si la partie déclarative et le corps sont structurellement identiques à ceux d'un programme principal, l'en-tête est différent. Il se compose du mot réservé procedure suivi du nom de la **procédure** (voir document "Présentation des programmes") et, optionnellement, de paramètres entre parenthèses (cf. 5.4).

En-tête de procédure




Liste incomplète de paramètres




**Exemples:**

```
procedure dessiner_carre;        en-tête de procédure sans paramètre
(* Cette procédure ... *)

procedure line (x, y : integer); en-tête de procédure avec paramètres
(* Cette procédure ... *)
```

***Attention à ne pas oublier le commentaire (but, description des paramètres... cf. document "Présentation des programmes") accompagnant chaque en-tête de procédure ou de fonction!***

[ Retour à la table des matières ]

## 5.3 **Déclaration et exécution des procédures**

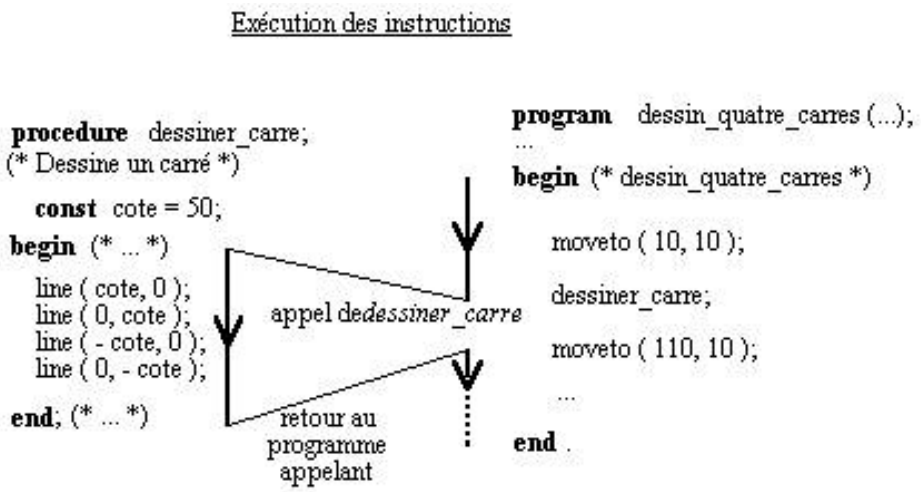
La déclaration des procédures intervient **après** celle des variables, juste avant le corps. Une procédure peut être déclarée dans la partie déclarative d'un programme principal mais aussi **à l'intérieur** d'une autre procédure, également dans sa partie déclarative. Ceci implique que la structure du programme et de ses procédures pourra refléter fidèlement la décomposition des problèmes par raffinements successifs (cf. 2.2)!

**Exemple:**

```
procedure externe (...);
   (* déclarations des constantes et variables de la procédure externe *)

   procedure interne (...);
      (* déclarations de la procédure interne *)

   begin (* interne *)
      ...
```

```
   end; (* interne *)

begin (* externe *)
   ...
end; (* externe *)
```

L'exécution des **instructions** d'une procédure, par contraction **exécution d'une procédure**, est commandée par l'exécution de **l'instruction d'appel** de la procédure. Le schéma qui suit, utilisant le programme *dessin_quatre_carres* (cf. 5.1), décrit le mécanisme utilisé:




[ Retour à la table des matières ]

## 5.4 **Paramètres des procédures**

### 5.4.1 **Paramètres d'entrée**

La procédure *dessiner_carre* est bien utile pour dessiner un carré. Elle le serait plus encore si elle permettait le dessin d'un carré de **n'importe quelle taille.** Pour cela il faudrait que la constante cote soit remplacée par une "variable" dont la valeur est donnée **au moment de l'appel** de la procédure.

Ce mécanisme existe! La "variable" est appelée **paramètre.** La déclaration des paramètres a lieu dans l'en-tête des procédures (cf. 5.2).

La procédure *dessiner_carre* devient:

```
procedure dessiner_carre (cote : integer);
(* Cette procédure dessine un carré dont la longueur du côté *)
(* est passée en paramètre *)

begin (* dessiner_carre *)
   line ( cote, 0 );  (* dessin d'un carré de côté cote. L'instruction *)
   line ( 0, cote );  (* line(dx,dy); dessine un trait depuis la position *)
   line ( -cote,0 );  (* courante (x,y) jusqu'au point (x+dx,y+dy) qui *)
   line ( 0, -cote ); (* devient la position courante. *)
end; (* dessiner_carre *)
```

L'instruction d'appel de la procédure est modifiée en spécifiant entre parenthèses la valeur (expression entière) qui doit être donnée au paramètre *cote*.

Il paraît naturel que le **type du paramètre** et le **type de la valeur** spécifiée doivent être **identiques**. Une seule exception: la valeur peut être entière et le paramètre réel (penser à l'affectation!).

**Exemples:**

```
   dessiner_carre ( 50 );                  le carré aura 50 unités de côté

   dessiner_carre ( nb_entier );            le carré aura nb_entier unités de côté

   dessiner_carre ( ( 10 + nombre ) div 3 ); le carré aura un côté égal à la valeur
de
                                            l'expression ( 10 + nombre ) div 3
```

**Remarques:**

1. Lesparamètres déclarés et utilisés dans les procédures sont appelés **paramètres formels**; les valeurs spécifiées à l'appel sont appelées **paramètres effectifs.**

2. Dans notre exemple la valeur représentant la longueur d'un côté est transmise de **l'extérieur vers l'intérieur** de la procédure. On parle dans ce cas de **paramètre d'entrée.**

3. N'importe quel type peut être utilisé dans une déclaration de paramètre.

4. Le lecteur malin aura remarqué que, par exemple, l'instruction (cf. 3.6.5)

   write ( 'Le resultat vaut:', 56 );

   est l'appel de la procédure *write* avec 2 paramètres d'entrée: '*Le resultat vaut*:' et 56.

[ Retour à la table des matières ]

### 5.4.2 **Paramètres de sortie**

Reprenons encore une fois la procédure *dessiner_carre* . Nous voudrions maintenant qu'elle calcule la surface du carré en plus de son dessin, et nous voulons utiliser cette surface à l'extérieur de la procédure.

Nous allons également utiliser le mécanisme des paramètres pour transmettre la valeur calculée. La procédure devient:

```
   procedure dessiner_carre (cote : integer;
                             var surface : integer);
   (* Cette procédure dessine un carré dont la longueur du côté *)
   (* est obtenue par paramètre. Elle calcule la surface, qui est ensuite *)
   (* transmise par paramètre à l'extérieur *)

   begin (* dessiner_carre *)
      line ( cote,0 );    (* dessin d'un carré de côté cote. L'instruction *)
      line ( 0,cote );    (* line(dx,dy); dessine un trait depuis la position *)
      line ( -cote,0 );   (* courante (x,y) jusqu'au point (x+dx,y+dy) qui *)
      line ( 0,-cote );   (* devient la position courante. *)

      (* calcul de la surface du carré *)
      surface := sqr ( cote );
   end; (* dessiner_carre *)
```

La déclaration du paramètre *surface* est précédée du mot réservé **var.** Cela signifie que *surface* permet de transmettre une valeur de **l'intérieur vers l'extérieur** de la procédure. Cette valeur est transmise **à la fin** de l'exécution de la procédure. Un tel paramètre est appelé **paramètre de sortie**.

Les paramètres effectifs doivent être ici des **variables** (et non des expressions) puisqu'ils vont recevoir une valeur. En effet:

- `dessiner_carre ( 50, aire );` est un appel correct de la procédure *dessiner_carre* ci-dessus (à condition que aire soit de type *integer*)
- `dessiner_carre ( 50, 30 );` est un appel ***incorrect*** de la procédure *dessiner_carre* ci-dessus (comment le nombre 30 pourrait-il "recevoir" une valeur?)

Ici également les paramètres formels et effectifs doivent être de (n'importe quel) même type.

Le lecteur malin aura remarqué que l'instruction

`read (x, y);` où *x* et *y* sont des variables réelles

est l'appel de la procédure prédéfinie *read* avec 2 paramètres de sortie qui recevront deux valeurs réelles données par l'utilisateur du programme.

[ Retour à la table des matières ]

### 5.4.3 **Paramètres d'entrée et de sortie**

Un paramètre d'entrée et de sortie est un paramètre permettant de transmettre des valeurs de l'extérieur vers l'intérieur d'une procédure **et inversément**.

Or en Pascal le mécanisme est le même que celui utilisé pour les paramètres de sortie (cf.5.4.2). Un paramètre dont la déclaration est précédée de **var** est aussi un paramètre d'entrée et de sortie. Cela signifie qu'il n'existe pas véritablement en Pascal de paramètres de sortie uniquement. Seule la logique de la procédure (ainsi que le commentaire mis dans l'en-tête!) indique si un tel paramètre est de sortie ou d'entrée et de sortie. Notons au passage que des langages comme Portal ou Ada possèdent les trois sortes de paramètres.

**Exemple:**

Considérons la procédure

```
procedure calculer_cube ( var nombre : integer );
  (* calcule le cube de la valeur passée en paramètre et livre cette valeur *)
  (* dans nombre *)

 begin (* calculer_cube *)
   nombre := nombre * nombre * nombre;
 end; (* calculer_cube *)
```

Le paramètre *nombre* est un paramètre d'entrée et de sortie: le calcul du cube utilise la valeur contenue dans *nombre* (transmise de l'extérieur), puis affecte le résultat au même paramètre *nombre*. Ce résultat est alors transmis à l'extérieur.

**Remarque**

Lorsqu'une procédure possède plusieurs paramètres ayant des modes de passage différents, il est conseillé de commencer par les paramètres d'entrée, puis de terminer par les paramètres de sortie chacun précédé du mot réservé **var**!

**Exemple:**

```
procedure exemple ( param_1 : real;
                    param_2 : boolean;
```

```
                var param_3 : integer;
                var param_4 : integer );
```

[ Retour à la table des matières ]

## 5.5 **Précisions sur le passage des paramètres**

### 5.5.1 **Passage par valeur**

Le passage d'une valeur à un paramètre non précédé du mot réservé **var** est appelé **passage par valeur**. En effet cette valeur est copiée dans le paramètre formel à l'appel de la procédure. Ceci signifie que, même si par la suite la valeur du paramètre formel est modifiée dans la procédure, la valeur du paramètre effectif **ne change pas**!

**Exemple:**

```
   program passage_par_valeur ( input, output );
      ...
      var lettre : char; (* contient une lettre minuscule pour notre exemple *)

      procedure majusculiser ( lettre_majuscule : char );
      (* Cette procédure transforme une lettre minuscule en *)
      (* lettre majuscule *)

      begin (* majusculiser *)
         lettre_majuscule := chr ( ord ( lettre_majuscule ) - ord ('a') + ord ('A')
);
         (* lettre_majuscule contient maintenant une valeur autre que celle *)
         (* passée en paramètre *)
        ...    (* autres instructions *)
      end; (* majusculiser *)

   begin (* passage_par_valeur *)
      lettre := 'b';
      majusculiser ( lettre );
      (* après exécution de majusculiser, lettre vaut toujours 'b' *)
      ...
   end.
```

L'appel suivant est également possible:

majusculiser ( 'b' );

[ Retour à la table des matières ]

### 5.5.2 **Passage par référence (ou par variable)**

Le passage d'une valeur à un paramètre précédé du mot réservé **var** est appelé **passage par référence** ou **par variable**. En fait ce passage n'est qu'une vue de l'esprit; il faut considérer que le paramètre formel n'est qu'un nouveau nom pour le paramètre effectif! Ceci signifie que, si par la suite la valeur du paramètre formel est modifiée, la valeur du paramètre effectif l'**est de la même manière!**

**Exemple:**

```
   program passage_par_référence ( input, output );
      ...
      var lettre : char;    (* contiendra une minuscule pour notre exemple *)
```

```
      procedure majusculiser (var lettre_majuscule : char);
      (* Cette procédure transforme une lettre minuscule en lettre majuscule *)

      begin (* majusculiser *)
         lettre_majuscule := chr( ord(lettre_majuscule) - ord('a') + ord('A') );
         (* lettre_majuscule contient maintenant une valeur autre que celle *)
         (* passée en paramètre *)
      end; (* majusculiser *)

   begin (* passage_par_référence *)
      lettre := 'b';
      majusculiser ( lettre );
      (* après exécution de majusculiser, lettre ne vaut plus 'b' mais la dernière *)
      (* valeur affectée au paramètre lettre_majuscule, à savoir la valeur 'B' *)
...
   end.
```

Il est donc clair que l'appel majusculiser ( 'b' ); est **faux.**

[ Retour à la table des matières ]

### 5.6 **Notions de bloc et de portée des identificateurs**

A la notion de procédure est associée celle de **bloc**. Un bloc est un nom donné à une région d'un programme contenant

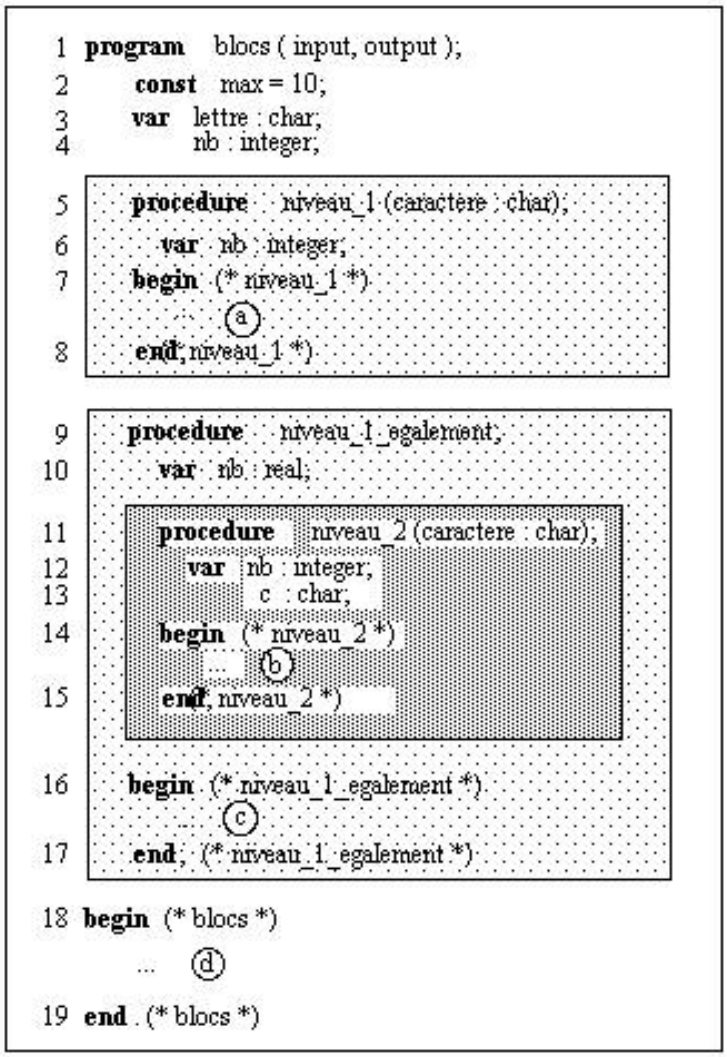
- une procédure
- une fonction
- le programme principal

Comme les procédures et les fonctions peuvent être emboîtées les unes dans les autres, on obtient des blocs également emboîtés. Cette structure appelée **structure de blocs** définit des niveaux de blocs:

1. Le programme principal forme le bloc de niveau 0
2. Les procédures et fonctions déclarées directement dans le programme principal forment chacune un bloc de niveau 1
3. Les procédures et fonctions déclarées dans des procédures ou fonctions de niveau 1 forment chacune un bloc de niveau 2
4. De manière générale: les procédures et fonctions déclarées dans des procédures ou fonctions de niveau N forment chacune un bloc de niveau N+1

Le numéro de niveau est appelé la **profondeur** du niveau. Un bloc de niveau 5 est plus profond qu'un bloc de niveau 2.

**Exemple:**




Cet exemple montre une structure de blocs à 3 niveaux: le programme principal appelé blocs forme le niveau 0, les procédures *niveau_1* et *niveau_1_egalement* constituent chacune un bloc et forment le niveau 1, finalement la procédure *niveau_2* forme un bloc de niveau 2.

Ces niveaux permettent d'introduire la notion de portée d'un identificateur. En effet un identificateur est connu et utilisable

- dans son bloc de définition dès que l'identificateur a été déclaré
- dans tous les **blocs de niveau plus profond,** s'ils sont déclarés **dans et après** son bloc de définition.

La région d'un programme dans laquelle peut être utilisé un identificateur est appelée **portée de l'identificateur.**

**Exemples**:

Considérons les identificateurs déclarés dans la figure ci-dessus. On obtient

```
- blocs              id. (nom du programme) unique
- max                id. (de constante) déclaré 1 fois
- lettre             id. (de variable) déclaré 1 fois
- nb                 id. (de variable) déclaré 4 fois
- c                  id. (de variable) déclaré 1 fois
- caractere          id. (de paramètre) déclaré 2 fois
- niveau_1           id. (de procédure) déclaré 1 fois
- niveau_1_egalement id. (de procédure) déclaré 1 fois
- niveau_2           id. (de procédure) déclaré 1 fois
```

Considérons à présent les régions a, b, c et d. Quels sont les identificateurs utilisables dans chaque région, et quels objets désignent-ils?

**Région a**: Les identificateurs **visibles** à cet endroit, c'est-à-dire ceux que le programmeur peut utiliser, sont

- *max*
- *lettre*
- *caractere* paramètre déclaré à la ligne 5
- *nb* qui désigne la variable déclarée à la ligne 6! Il est donc impossible d'utiliser l'identificateur nb déclaré à la ligne 4; celui-ci est dit **caché**.
- *niveau_1* (appel récursif direct, cf.11.2.2 )

**Région b**: Les identificateurs **visibles** à cet endroit, c'est-à-dire ceux que le programmeur peut utiliser, sont

- *max*
- *lettre*
- *caractere* paramètre déclaré à la ligne 11
- *nb* qui désigne la variable déclarée à la ligne 12! Les identificateurs *nb* déclarés aux lignes 4 et 10 sont **cachés**.
- *c*
- *niveau_1*
- *niveau_2* (appel récursif direct, cf. 11.2.2)
- *niveau_1_egalement* (appel récursif indirect, cf. 11.4)

**Région c:** Les identificateurs **visibles** à cet endroit, c'est-à-dire ceux que le programmeur peut utiliser, sont

- *max*
- *lettre*
- *nb* qui désigne la variable déclarée à la ligne 10! L'identificateur *nb* déclaré à la ligne 4 est **caché.**
- *niveau_1*
- niveau_2
- *niveau_1_egalement* (appel récursif direct, cf. 11.2.2)

**Région d**: Les identificateurs **visibles** à cet endroit, c'est-à-dire ceux que le programmeur peut utiliser, sont

- *max*
- *lettre*
- *nb* qui désigne la variable déclarée à la ligne 4.
- *niveau_1*
- *niveau_1_egalement*

**Remarque importante**

Une conséquence de tout ceci est que plusieurs objets déclarés dans des blocs différents peuvent porter le même nom sans que l'on risque de les confondre!

[ Retour à la table des matières ]

## 5.7 **Identificateurs locaux et globaux**

Les deux définitions suivantes s'emploient également pour qualifier un identificateur:

- Un identificateur est **local** à un bloc s'il est déclaré dans la partie déclarative du bloc.

- Un identificateur est **global** à un bloc s'il est déclaré à l'extérieur du bloc et que le bloc fait partie de la portée de cet identificateur.

**Exemples:**

1. *max* est local au programme principal et global à toutes les procédures

2. caractere déclaré à la ligne 5 est local à la procédure *niveau_1; caractere* déclaré à la ligne 12 est local à la procédure *niveau_2*

3. *niveau_1* est local au programme principal; il est global aux procédures *niveau_1_egalement* et *niveau_2*

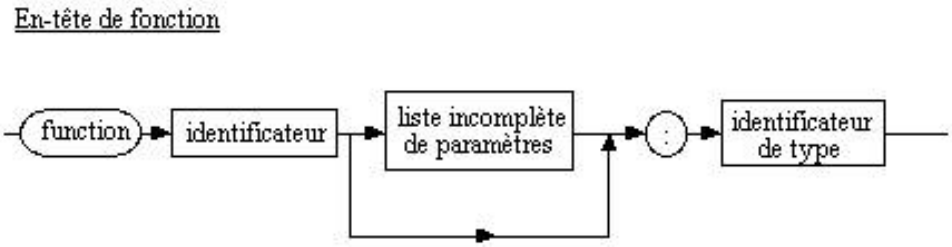
Mentionnons encore au lecteur un peu effrayé par cette terminologie que les notions ainsi définies sont conformes à une certaine logique. Avec un peu d'habitude toutes ces explications paraissent même partiellement superflues.

[ Retour à la table des matières ]

## 5.8 **Les fonctions**

Une fonction est également une construction permettant de structurer les programmes. Tout ce qui a été dit pour les procédures est valable pour les fonctions, aux différences suivantes près:

1. Une fonction a pour but de **calculer une valeur** appelée **résultat** de la fonction. L'en-tête d'une fonction comprend le mot réservé function suivi du nom de la **fonction** (voir document "Présentation des programmes") et, optionnellement, de paramètres (cf. 5.4), puis du **type** du résultat.




   **Exemples:**

   - **function** aleatoire : real;

   - **function** sinus ( angle_en_radians : real ) : real;

   - **function** puissance ( nombre : real;
                              exposant : integer ) : real;

   Précisons que le type du résultat doit être un type **simple** (cf. annexe F) ou un type pointeur (cf. 9.3). De nombreux compilateurs admettent cependant des extensions.

2. L'appel d'une fonction n'est pas une instruction (contrairement à l'appel de procédure) mais fait partie d'une **expression**. Il consiste à nommer la fonction et à donner la liste des paramètres effectifs entre parenthèses si nécessaire.

   **Exemples:**

   - variable_reelle := aleatoire;

- cote_oppose := sinus ( pi / 4.0 ) * hypothenuse; (* pi est une constante réelle *)

- **if** exposant >= 0 **then**
  variable_reelle := puissance ( variable_reelle, exposant );

3. Le résultat d'une fonction est obtenu grâce à son **nom**. Pour cela, le nom de la fonction est considéré comme une variable locale spéciale utilisable dans son corps. Cette **variable spéciale** ne peut cependant qu'être affectée (la consultation équivaudrait à une tentative d'appel récursif, cf. 11.2). Le résultat de la fonction sera donc la valeur contenue dans cette variable à la fin de l'exécution de la fonction.

**Exemple:**

```
    function puissance ( nombre : real;
                                          exposant : integer) : real;
     (* Cette fonction calcule la puissance d'un nombre réel *)

                 var i : integer;                (* variable de boucle *)
                       resultat : real;          (* produits intermédiaires *)

      begin (* puissance *)
              resultat := 1.0;
              for i := 1 to exposant do                   (* multiplier le
nombre exposant  fois par *)
                        resultat := resultat * nombre;        (* lui-même *)

               puissance := resultat;           (* pour que la fonction redonne
la valeur *)
                                                (* contenue dans resultat comme
résultat *)
       end; (* puissance *)
```

**Remarque importante**

Les paramètres d'une fonction devraient toujours être des paramètres d'**entrée.** Il existe quelques cas ***exceptionnels*** où ceux-ci sont des paramètres d'entrée et de sortie (cas d'un paramètre fichier par exemple, cf.10.3.1).

[ Retour à la table des matières ]

## 5.9 **Procédures et fonctions prédéfinies**

Pascal, comme tout langage de programmation, possède des procédures et fonctions prédéfinies. Celles-ci sont donc utilisables directement sans faire l'objet d'une déclaration explicite.

Citons par exemple les procédures read, readln, write, writeln et les fonctions *abs, sqrt, trunc, sin* etc (cf. annexe E).

[ Retour à la table des matières ]

## 5.10 **Procédures et fonctions passées comme paramètres**

Il est possible de passer des procédures et des fonctions comme paramètres. Ceci fait l'objet d'un paragraphe spécial (cf. 12.3), les mécanismes utilisés étant syntaxiquement semblables mais sémantiquement très différents.

[ Retour à la table des matières ]

**CHAPITRE 11**

# LA RECURSIVITE

## 11.1 **Remarques préliminaires**

Ce chapitre va présenter non pas une nouvelle structure du langage Pascal mais une technique permettant l'écriture d'algorithmes particuliers appelés algorithmes récursifs (cf. ci-dessous). Précisons tout de suite que seules quelques idées générales seront données et non une théorie complète!

[ Retour à la table des matières ]

## 11.2 **Exemple classique**

### 11.2.1 **Présentation du problème**

On définit une suite de nombres tels que le n-ième élément de cette suite soit égal à la somme des deux précédents. On décide que les deux premiers nombres sont 0 et 1. La suite obtenue est la suivante:

0 1 1 2 3 5 8 13 21 34 55 ...

Formellement, si le n-ième nombre est noté Fn, tout élément se calcule par la formule

Fn = Fn-1 + Fn-2

Une telle formule est dite **récursive** car le calcul de Fn implique la connaissance et l'utilisation de Fn-1 et Fn-2 qui, eux-mêmes, impliquent celles de Fn-3 et Fn-4 etc.

Notons que les Fi sont appelés nombres de Fibonacci.

**Définition**

De manière générale la **récursivité** est le fait de définir un objet, une action... au moyen de cet objet, de cette action...

[ Retour à la table des matières ]

### 11.2.2 **Application au langage Pascal**

Pascal permet qu'une procédure ou une fonction s'appelle elle-même; une telle procédure ou fonction est appelée récursive.

Dans le cas des nombres de Fibonacci une fonction fournissant le nombre cherché (et non une procédure) s'impose. L'en-tête de cette fonction sera

```
function fibonacci ( n : integer ) : integer;
 (* calcule le n-ième nombre de Fibonacci, n étant donné en paramètre *)
```

Comment écrire le corps de la fonction? Il faut considérer trois cas:

1. n = 0 la fonction donne 0 comme résultat
2. n = 1 la fonction donne 1 comme résultat
3. autre n la fonction donne Fn-1 + Fn-2 comme résultat

On obtient donc la fonction suivante:

```
function fibonacci ( n : integer ) : integer;
(* calcule le nème nombre de Fibonacci, n étant donné en paramètre *)

begin (* fibonacci *)

  if n = 0 then           (* cas de F0 *)
       fibonacci := 0

  else if n = 1 then     (* cas de F1 *)
             fibonacci := 1

        else             (* tous les autres cas *)
             fibonacci := fibonacci ( n - 1 ) + fibonacci ( n - 2 ); (* appels
récursifs *)

end; (* fibonacci *)
```

**Question à méditer**

Combien d'appels de la fonction fibonacci effectue un programme désirant calculer F3 ? F4 ? F5 ?

[ Retour à la table des matières ]

## 11.3 **Mise en oeuvre de la récursivité**

Etant donné un problème (peut-être de nature récursive) comment construire un algorithme récursif de résolution?

Plusieurs points doivent être considérés:

1. Décomposer le problème en sous-problèmes (cf. 2.2); ceci fixera l'algorithme et permettra de vérifier que le problème peut être résolu de manière récursive.

2. Déterminer quelles sont les données nécessaires à l'obtention de la solution du problème.

3. Trouver une (des) condition(s) de terminaison de l'algorithme. Ceci se fait en considérant un (des) cas simple(s) où l'algorithme donne la solution sans appels récursifs. **Tout algorithme récursif doit être fini!**

Illustrons tout cela par un exemple:

> Soit une liste chaînée par pointeurs (cf. 9.4.2). Chaque élément de cette liste contient un nombre entier. Ecrire une procédure donnant la somme des nombres de la liste.

Les structures de données nécessaires à la liste sont

```
type lien = ^element;     (* pour le chaînage, convention t_ non utilisée ici *)
     element = record
                  nombre : integer; (* nombre à considérer *)
                  suivant : lien;   (* lien de chaînage *)
               end;
```

```
var  tête_de_liste : lien;   (* pointe sur le premier élément de la liste *)
     somme_finale : integer; (* somme des nombres *)
```

Décomposons le problème en sous-problèmes:

a) considérer l'élément courant de la liste
b) additionner sa valeur à la somme des valeurs des éléments déjà parcourus
c) passer à l'élément suivant de la liste

On voit alors que les données nécessaires au calcul de la somme sont

- le nombre contenu dans chaque élément (évident)
- les sommes partielles
- le pointeur repérant l'élément courant

Ecrivons partiellement la procédure demandée:

```
procedure parcours_somme  ((* paramètres à définir *)) ;
(* parcourt la liste récursivement en additionnant les valeurs des éléments *)

begin (* parcours_somme *)

  (* considérer l'élément courant *)

  (* additionner la valeur à la somme des valeurs des éléments déjà parcourus *)
  somme := somme + courant^.nombre;

  (* passer à l'élément suivant *)
  parcours_somme ( (* paramètres effectifs *) );

end; (* parcours_somme *)
```

On voit maintenant que les paramètres de la procédure sont la somme partielle ainsi qu'un pointeur sur l'élément courant. Donc

```
procedure parcours_somme ( courant : lien;
                           var somme : integer );
(* parcourt la liste récursivement en additionnant les valeurs des éléments *)

begin (* parcours_somme *)

  (* considérer l'élément courant *)

  (* additionner la valeur à la somme des valeurs des éléments déjà parcourus *)
  somme := somme + courant^.nombre;

  (* passer à l'élément suivant *)
  parcours_somme ( courant^.suivant, somme );

end; (* parcours_somme *)
```

Il reste à trouver une condition de terminaison de l'algorithme: le parcours est terminé lorsque la fin de la liste est atteinte, ce qui s'exprime au moyen du pointeur courant. Finalement

```
procedure parcours_somme ( courant : lien;
                           var somme : integer );
(* parcourt la liste récursivement en additionnant les valeurs des éléments *)
```

```
begin (* parcours_somme *)

  (* considérer l'élément courant: stop si la fin de liste est atteinte *)
  if courant <> nil then
  begin

    (* additionner la valeur à la somme des valeurs des éléments déjà parcourus *)
    somme := somme + courant^.nombre;

    (* passer à l'élément suivant *)
    parcours_somme ( courant^.suivant, somme );

  end;

end; (* parcours_somme *)
```

**Remarques:**

1. L'appel de cette procédure se fait par

   *parcours_somme ( tete_de_liste, somme_finale );*

   avec *somme_finale* initialisée à 0 avant!
2. Cette procédure se comporte normalement même si la liste est vide, c'est-à-dire si *tete_de_liste* vaut **nil** .

[ Retour à la table des matières ]

## 11.4 **Récursivité croisée**

Les procédures récursives vues jusqu'à présent s'appellent elles-mêmes directement. Que se passe-t-il si une procédure P1 appelle une procédure P2 qui appelle P1? Ce cas est un cas de récursivité **croisée** (ou indirecte).

**Exemple:**

```
procedure parcours_somme ( courant : lien;
                           var somme : integer );
(* parcourt la liste récursivement en additionnant les valeurs des éléments *)

  procedure somme ( courant : lien;
                    var somme : integer );
  (* additionne la valeur courante et passe à l'élément suivant. Illustre *)
  (* la récursivité croisée *)

  begin (* somme *)

    (* additionner la valeur à la somme des valeurs des éléments déjà parcourus *)
    somme := somme + courant^.nombre;

    (* passer à l'élément suivant *)
    parcours_somme ( courant^.suivant, somme ); (* appel récursif croisé *)

  end; (* somme *)

begin (* parcours_somme *)

  (* considérer l'élément courant: stop si la fin de liste est atteinte *)
```

```
    if courant <> nil then
       somme ( courant, somme );              (* appel récursif croisé *)

  end; (* parcours_somme *)
```

Cette solution a le désavantage de rendre la procédure *somme* interne à *parcours_somme* donc inutilisable ailleurs que dans cette dernière. Comment faire pour déclarer ces deux procédures au même niveau? Cela pose en effet un problème: la déclaration de *somme* doit précéder celle de *parcours_somme* puisque celle-ci est appelée par *somme* et inversément!

Pour résoudre ce problème Pascal met à disposition le mot réservé **forward**. Celui-ci permet d'écrire l'en-tête d'une procédure (ou fonction) seule, la déclaration principale ayant lieu plus loin dans la même partie déclarative. **forward** se place après la déclaration de l'en-tête et est suivi d'un point-virgule.

ATTENTION: Lors de la déclaration principale l'en-tête comporte uniquement **procedure** ou **function** suivi du nom de celle-ci!

**Exemple:**

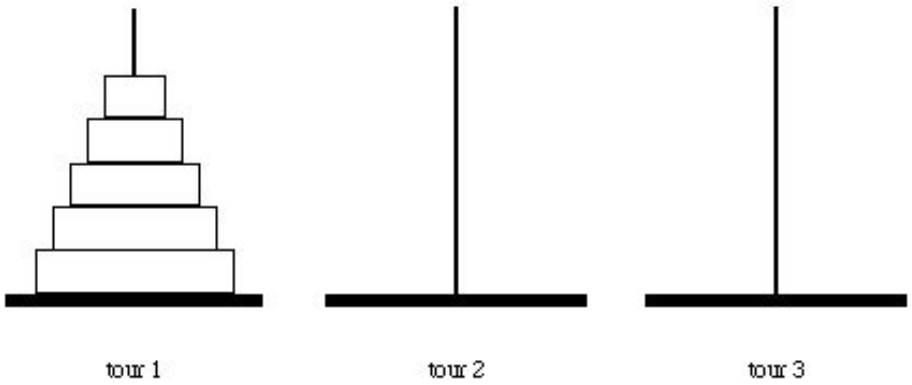
```
  procedure somme ( courant : lien;
                    var somme : integer ); forward;
  (* Le mot réservé forward indique que la procédure somme est déclarée *)
  (* plus loin (dans la même partie déclarative). Ne pas oublier les ";" *)


  procedure parcours_somme ( courant : lien;
                             var somme : integer );
  (* parcourt la liste récursivement en additionnant les valeurs des éléments *)

  begin (* parcours_somme *)

    (* considérer l'élément courant: stop si la fin de liste est atteinte *)
    if courant <> nil then
       somme ( courant, somme ); (* appel récursif croisé *)

  end; (* parcours_somme *)


  (* Déclaration principale de somme. Les paramètres sont mis en commentaire *)
  (* pour des raisons de lisibilité de cette déclaration! *)

  procedure somme;       (* courant : lien; *)
                          (* var somme : integer *)

  (* additionne la valeur courante et passe à l'élément suivant. Illustre *)
  (* la récursivité croisée *)

  begin (* somme *)

    (* additionner la valeur à la somme des valeurs des éléments déjà parcourus *)
    somme := somme + courant^.nombre;

    (* passer à l'élément suivant *)
    parcours_somme ( courant^.suivant, somme ); (* appel récursif croisé *)

  end; (* somme *)
```

[ Retour à la table des matières ]

## 11.5 **Les tours de Hanoï**

Voici un jeu célèbre dont la simulation sur ordinateur fait appel à la récursivité:

Considérons trois tours numérotées de 1 à 3. Sur la tour 1 sont posés des disques de manière à constituer une pyramide:




Le jeu consiste à déplacer tous les disques de la tour 1 sur la tour 2 en utilisant la tour 3 et en respectant la règle suivante: on ne peut transférer le disque du sommet d'une tour que sur un disque plus grand ou sur un socle vide!

Commençons par décomposer ce problème en sous-problèmes:

Soit N le nombre de disques sur la tour 1. Il faut alors:

- transférer N-1 disques de la tour 1 sur la tour 3 en utilisant la tour 2 comme intermédiaire

- transférer le disque restant de la tour 1 sur le socle de la tour 2

- transférer les N-1 disques de la tour 3 sur la tour 2 en utilisant la tour 1 comme intermédiaire.

On obtient donc le squelette de procédure suivant:

```
procedure deplacer ( (* paramètres *) );

begin (* deplacer *)
  (* transférer N-1 disques sur une tour vide *)
  deplacer ( ... );

  (* transférer le disque restant *)
  writeln ('Deplacer le disque restant sur la tour devenue vide');

  (* transférer N-1 disques sur celui qui vient d'être déplacé *)
  deplacer ( ... );
end; (* deplacer *)
```

Les paramètres de cette procédure sont les données nécessaires à son exécution, à savoir

- le nombre de disques à transférer

- les tours d'origine, intermédiaire et de destination

On obtient donc:

```
procedure deplacer ( nb_disques : integer;
                     tour_origine : integer;
                     tour_destination : integer;
                     tour_intermediaire : integer );
(* cette procédure transfère nb_disques depuis tour_origine sur tour_destination *)
(* en utilisant tour_intermediaire comme intermédiaire *)

begin (* deplacer *)

  (* transférer nb_disques - 1 disques sur tour_intermediaire en utilisant *)
  (* tour_destination comme intermédiaire *)
  deplacer ( nb_disques - 1, tour_origine, tour_intermediaire, tour_destination );

  (* transférer le disque restant de tour_origine sur tour_destination. On *)
  (* représente cela par l'affichage d'un message à l'écran *)
  write ('Deplacer le disque restant de la tour', tour_origine : 2);
  writeln (' sur la tour', tour_destination : 2);

  (* transférer nb_disques - 1 disques sur tour_destination en utilisant *)
  (* tour_origine comme intermédiaire *)
  deplacer ( nb_disques - 1, tour_intermediaire, tour_destination, tour_origine );

end; (* deplacer *)
```

Reste à trouver un cas de terminaison de l'algorithme: lorsqu'il faut transférer 0 disques! On obtient donc finalement:

```
procedure deplacer ( nb_disques : integer;
                     tour_origine : integer;
                     tour_destination : integer;
                     tour_intermediaire : integer );
(* cette procédure transfère nb_disques depuis tour_origine sur tour_destination *)
(* en utilisant tour_intermediaire comme intermédiaire *)

begin (* deplacer *)

  if nb_disques > 0 then       (* encore des disques à transférer? *)
  begin
    (* transférer nb_disques - 1 disques sur tour_intermediaire en utilisant *)
    (* tour_destination comme intermédiaire *)
    deplacer ( nb_disques - 1, tour_origine, tour_intermediaire, tour_destination );

    (* transférer le disque restant de tour_origine sur tour_destination. On *)
    (* représente cela par l'affichage d'un message à l'écran *)
    write ('Deplacer le disque restant de la tour', tour_origine : 2);
    writeln (' sur la tour', tour_destination : 2);

    (* transférer nb_disques - 1 disques sur tour_destination en utilisant *)
    (* tour_origine comme intermédiaire *)
    deplacer ( nb_disques - 1, tour_intermediaire, tour_destination, tour_origine );
  end;

end; (* deplacer *)
```

[ Retour à la table des matières ]

## 11.6 **Suppression de la récursivité**

### 11.6.1 **Motivation**

Supposons qu'un programmeur malchanceux doit réaliser la procédure deplacer ci-dessus dans un langage ne permettant pas la récursivité (Fortran, Cobol ...). Comment doit-il faire?

Une bonne méthode est de supposer que Pascal ne permet pas la récursivité; il doit donc écrire une procédure non récursive qu'il traduira ensuite dans le langage à utiliser. Le problème qui va nous intéresser est évidemment la suppression de la récursivité.

[ Retour à la table des matières ]

### 11.6.2 **Comment fonctionne un appel récursif**

Lorsqu'un sous-programme s'appelle récursivement, son exécution est suspendue à l'endroit (du code) où l'appel a lieu; cet appel récursif provoque l'exécution de ce sous-programme dès le début de son code avec de **nouvelles valeurs pour les objets locaux** au sous-programme. Lorsque cette exécution se termine l'exécution reprend à l'instruction suivant l'appel récursif. Donc les objets suivants:

- paramètres du sous-programme
- variables locales du sous-programme
- résultat actuel de la fonction si ce sous-programme est une fonction
- adresse de reprise de l'exécution (instruction suivant l'appel récursif)

sont sauvés lors de l'appel récursif et réutilisés lorsque cet appel se termine.

**Exemple:**

Soit la procédure

```
procedure exemple ( nombre : integer );
(* destinée à illustrer ce qu'est un appel récursif *)

  var resultat : integer;

begin (* exemple *)

  resultat := nombre + 1;
  if resultat < 0 then      (* condition de terminaison *)
        exemple ( resultat );
  writeln ( resultat );

end; (* exemple *)
```

Que fait cette procédure si on l'appelle avec  exemple ( -2 ); ?

Premièrement l'effet visible sera l'affichage suivant:

```
 0
-1
```

Deuxièmement que se passe-t-il plus précisément?

1. L'appel initial ( *exemple*  ( -2 ); ) crée une zone mémoire contenant les valeurs de *nombre*  et *resultat*  à savoir -2 et *indéfini* ainsi que l'adresse de reprise utilisée pour retourner au programme appelant. Appelons Z1 cette zone mémoire.

2. L'instruction *resultat := nombre + 1*; fait que Z1 contient les valeurs -2 et -1.

3. L'instruction **if** est exécutée et l'appel récursif a lieu:

   - Z1 devient inaccessible.
   - une nouvelle zone Z2 est créée en mémoire. Z2 contient les valeurs des nouveaux *nombre* et *resultat* à savoir -1 et *indéfini* ainsi que l'adresse de reprise, c'est-à-dire l'adresse de l'instruction suivante à exécuter ( *writeln ( resultat );* ).
   - l'exécution reprend au début du code de la procédure *exemple* .

4. L'instruction *resultat := nombre + 1* ; fait que Z2 contient les valeurs -1 et 0.

5. L'instruction if n'est pas exécutée (resultat vaut 0).

6. L'instruction writeln ( resultat ); est exécutée (affichage de la valeur 0)

7. L'appel récursif se termine:

   - l'exécution reprend à l'instruction dont l'adresse est contenue dans Z2 ( adresse de reprise ) à savoir *writeln ( resultat );* ( voir point 3 ).
   - Z2 est détruite et Z1 redevient accessible.

8. L'instruction *writeln* ( *resultat* ); est exécutée (affichage de la valeur -1)

9. L'appel initial exemple ( -2 ); se termine.

On peut dire que les zones Z1 et Z2 sont empilées. L'accès aux objets *nombre* et *resultat* se fait toujours dans la zone située au sommet de la pile!

[ Retour à la table des matières ]

### 11.6.3 **Suppression d'appels récursifs**

Pour supprimer un appel récursif de sous-programme, il faut simuler ce qui est exécuté lors de l'appel et à la fin de cet appel. Il faut donc

- sauver les valeurs des paramètres d'entrées ( en fait des variables locales les représentant ) et des variables locales ceci au moyen d'une pile
- donner les nouvelles valeurs à ces paramètres
- sauveur l'adresse de reprise
- effectuer un saut au déébut du sous-programme

A la fin de l'appel il faut

- restituer les valeurs des paramètres d'entrée
- restituer les valeurs des variables locales
- sauter à l'adresse de reprise

A titre d'exemple voici une version dérécursifiée des tours de Hanoï. Avant d'examiner l'algorithme il faut préciser que

- les sauts aux adresses de reprise sont remplacés par les structures de contrôles habituelles.

- comme il n'y a que deux appels récursifs l'indicateur de l'adresse de reprise ne prend que les valeurs 0 et 1. La pile des adresses peut alors être représentée par un entier *( pile )* initialisé à 1.

- l'empilement des valeurs 0 et 1 sera simulé par les opérations

```
pile := pile * 2;                (* empiler 0 *)
pile := pile * 2 + 1;            (* empiler 1 *)
```

- pour connaître la valeur du sommet de pile on utilise *odd ( pile )*

- la suppression de l'élément au sommet de pile se fait par *pile := pile* ***div*** *2;*

Voici la procédure:

```
procedure deplacer ( nb_disques : integer;
                     tour_origine : integer;
                     tour_destination : integer;
                     tour_intermediaire : integer );
(* cette procédure transfère nb_disques depuis tour_origine sur tour_destination *)
(* en utilisant tour_intermediaire comme intermédiaire. Version non recursive. *)

   var pile : integer; (* simule la pile *)

   procedure echange ( var nombre_1, nombre_2 : integer );
   (* permute les valeurs de nombre_1 et nombre_2 *)

      var inter : integer; (* variable intermédiaire *)

   begin (* echange *)
      inter := nombre_1;
      nombre_1 := nombre_2;
      nombre_2 := inter;
   end; (* echange *)

begin (* deplacer *)

   pile := 1;
   repeat

      while nb_disques > 0 do       (* 1er appel récursif *)
      begin
         pile := pile * 2 + 1;
         echange ( tour_destination, tour_intermediaire );
         nb_disques := nb_disques - 1;
      end;

      while not odd ( pile ) do     (* 2ème retour *)
      begin
         echange ( tour_origine, tour_intermediaire );
         nb_disques := nb_disques + 1;
         pile := pile div 2;
      end;

      echange ( tour_destination, tout_intermediaire ); (* 1er retour *)
```

```
         nb_disques := nb_disques + 1;
         pile := pile div 2;

         if pile <> 0 then
         begin
            write ( 'Deplacer le disque restant de la tour', tour_origine : 2 );
            writeln ( ' sur la tour', tour_destination : 2 );
            pile := pile * 2;                          (* 2ème appel récursif *)
            echange ( tour_destination, tour_intermediaire );
            nb_disques := nb_disques - 1;
         end;

      until pile = 0;

   end; (* deplacer *)
```

[ Retour à la table des matières ]

**CHAPITRE 9**

# LES TYPES POINTEUR

## 9.1 **Motivation**

Jusqu'à présent la représentation d'un groupe de données de même nature (même type) n'est possible qu'au moyen des **tableaux** (cf. 6.3). Or leur facilité d'utilisation peut être fortement réduite lorsque:

a) la structure linéaire d'un tableau ("un élément après l'autre") ne reflète pas l'organisation globale des données comme par exemple

- les liaisons ferroviaires ou routières entre les différentes gares ou villes d'une région;
- toute hiérarchie de notre bas monde.

En effet un tableau peut difficilement refléter les successeurs ou prédécesseurs d'une donnée, les relations existant entre certaines données...

b) le nombre maximal d'éléments du tableau n'est **pas connu à la compilation** (ne peut pas être fixé par le programmeur), à savoir

- si ce nombre est choisi trop petit le programme ne pourra représenter et traiter toutes les données;
- s'il est pris (beaucoup) trop grand le gaspillage de mémoire peut pénaliser non seulement le fonctionnement du programme mais aussi celui du système informatique entier.

Il faut alors disposer de structures plus générales permettant de représenter le nombre exact de données, nombre **connu à l'exécution du programme seulement,** ainsi que les relations entre elles.

Il est naturel d'utiliser une structure d'enregistrement pour **une** donnée. Les relations (liens) seront implantées au moyen de **pointeurs.**

[ Retour à la table des matières ]

## 9.2 **Utilisation d'un type pointeur sur un exemple**

Soit une ville représentée par l'enregistrement suivant

```
type t_ville = record
                  nom : t_nom_ville;      (* t_nom_ville est un type chaîne de *)
                                          (* caractères *)
                  nb_habitants : integer; (* nombre d'habitants de la ville *)
               end;
```

On veut implanter le lien que constitue la liaison ferroviaire entre deux villes, par exemple Lausanne et Yverdon. On peut alors modifier la déclaration de type de la manière suivante:

```
type t_lien_ville = ^t_ville;                (* c'est un "type pointeur sur le
```

```
type t_ville" *)

        t_ville = record
                    nom : t_nom_ville;          (* t_nom_ville est un type chaîne de
*)
                                                (* caractères *)
                    nb_habitants : integer;     (* nombre d'habitants de la ville *)
                    ville_reliee : t_lien_ville;(* c'est un pointeur sur une ville
*)
                  end;

   var premiere_ville : t_lien_ville;           (* la première ville créée *)
       ville_courante : t_lien_ville;            (* pour créer une nouvelle ville *)
```

Grâce à ces deux variables appelées **pointeurs**, il va maintenant être possible d'obtenir à l'exécution du programme autant de villes (de type *t_ville* ) que nécessaire et souhaité (jusqu'à utilisation complète de la mémoire de la machine, naturellement!) et de travailler avec elles.

Les deux variables *premiere_ville* et *ville_courante* (de type *t_lien_ville* ), comme toutes les variables utilisées jusqu'à maintenant, sont appelées **statiques**, car elles sont déclarées par le programmeur et connues (à la compilation) par le compilateur, qui leur attribue un emplacement mémoire, comme à n'importe quelle autre variable d'ailleurs.

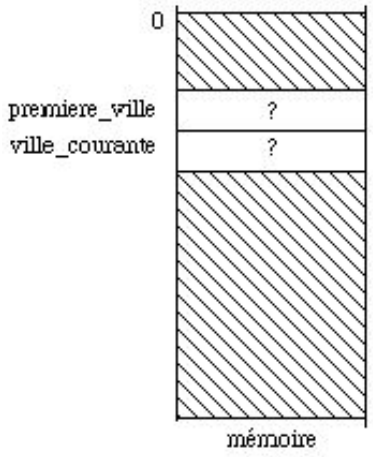
Or la particularité principale d'un pointeur réside dans son utilisation, principalement lorsqu'on le passe comme paramètre de la procédure prédéfinie *new* :

```
   new ( ville_courante );    (* c'est une instruction, bien entendu *)
```

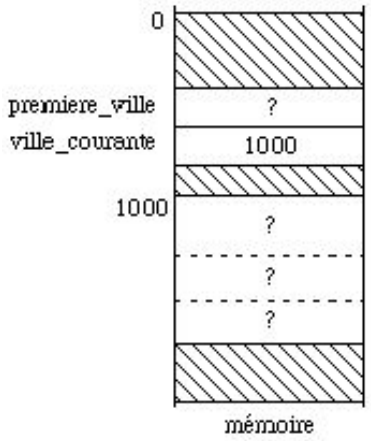
En effet, l'exécution de cette instruction provoque la **réservation**, dans la mémoire de l'ordinateur, d'une zone bien précise, juste suffisamment grande pour contenir une valeur du type *t_ville* (à savoir un nom, un nombre d'habitants et un pointeur).

Schématiquement

- avant l'exécution de l'instruction new ( *ville_courante* ); on avait:




- après l'exécution de l'instruction new ( *ville_courante* ); on obtient:




On remarque que la zone réservée (naturellement non initialisée) est "repérée", accessible grâce au pointeur *ville_courante* qui contient l'adresse mémoire (ici 1000 par exemple) de cette zone. Dans la syntaxe de Pascal, on la désignera par le nom de la variable pointeur suivi du chapeau, à savoir *ville_courante^* .

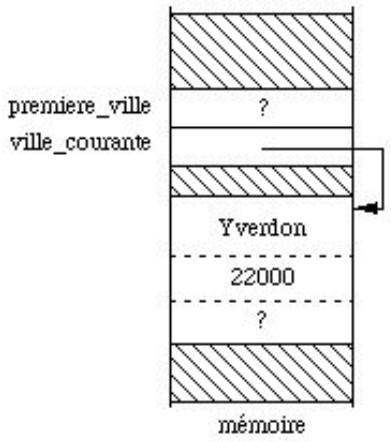
**Remarque**

Une telle zone est en fait appelée **variable dynamique**, car son contenu peut varier et parce qu'elle a été créée par **l'exécution d'une instruction.**

Il est maintenant possible de donner des valeurs aux champs *nom* et *nb_habitants* ainsi:

```
ville_courante^.nom := 'Yverdon';
ville_courante^.nb_habitants := 22000;
```

En effet, la variable dynamique ville_courante^ est de type *t_ville!!!* On obtient en mémoire:

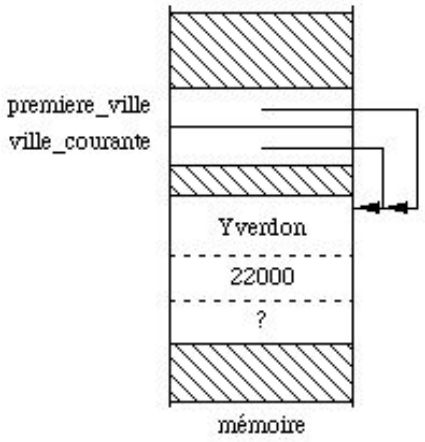



Il faut préciser que l'on ne s'intéresse absolument pas aux valeurs précises des adresses. C'est pour cette raison qu'on schématise avec une **flèche** que l'adresse de la variable *dynamique ville_courante^* est contenue dans la variable (statique) *ville_courante*.

Essayons maintenant de créer une deuxième ville (on décide arbitrairement qu'Yverdon est la première). Il faut alors repérer Yverdon grâce au pointeur *premiere_ville*, afin que *ville_courante* puisse repérer la deuxième. Cela se fait par une simple affectation:

```
premiere_ville := ville_courante;
```

En effet, cette affectation copie le contenu de *ville_courante* (en fait l'adresse de la variable dynamique) dans *premiere_ville*. Schématiquement, cela donne:



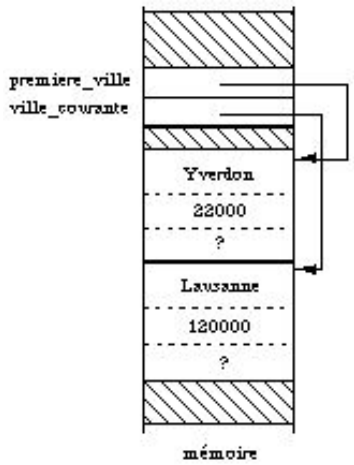

Puis on peut, une deuxième fois, exécuter l'instruction:

```
new ( ville_courante );
```

On va ainsi réserver une deuxième zone (une deuxième variable dynamique), repérée par *ville_courante*. Puis remplir les deux champs *nom* et *nb_habitants*:

```
ville_courante^.nom := 'Lausanne';
ville_courante^.nb_habitants := 120000;
```

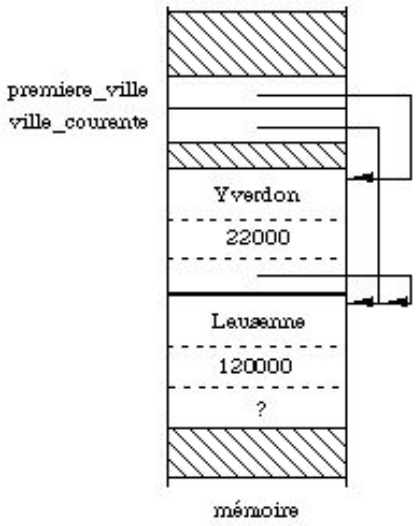
Schématiquement:




Comment faire maintenant pour créer une troisième ville? Comme on l'aura peut-être deviné, on va utiliser le champ *ville_reliee* pour relier les deux variables dynamiques entre elles, ce qui va permettre de réutiliser le pointeur *ville_courante* tout en gardant la possibilité de travailler avec les deux villes existantes. Cela se fait simplement par l'affectation:

```
premiere_ville^.ville_reliee := ville_courante;
```

Cette affectation est possible puisque le champ *ville_reliee* et la variable *ville_courante* sont du même type *t_lien_ville* !!! Schématiquement, cela donne:




On remarque immédiatement sur le dessin précédent que le pointeur *ville_courante* peut effectivement être réutilisé puisqu'on conserve un accès vers la deuxième ville (via la première).

Sans développer plus avant cet exemple (une solution complète est donnée à la fin de ce chapitre), on peut simplement remarquer que:

1. Avec deux variables **statiques** (*premiere_ville et ville_courante* ), il est maintenant possible d'implanter plusieurs autres **variables dynamiques** reliées les unes aux autres par des champs d'un type pointeur, ce qui résoud le problème du dimensionnement d'un groupe de données (cf. 9.1).

2. Le problème de l'organisation globale des données (cf. 9.1) est abordé mais non encore résolu complètement. Pour cela il faudrait relier une ville à plusieurs autres villes directement. Les pointeurs permettent également de réaliser de telles liaisons mais le montrer dépasserait le cadre de ce cours d'introduction.

[ Retour à la table des matières ]

## 9.3 **Les types pointeur**

### 9.3.1 **Généralités**

La déclaration d'un type pointeur est particulière: elle s'effectue au moyen du symbole ^ suivi d'un identificateur de type quelconque. Cet identificateur peut être déclaré **après** le type pointeur!!! **Exemples:**

```
   type t_lien_ville = ^t_ville;  (* cf. 9.2 *)
        pt_integer = ^integer;    (* peu utile, autant utiliser l'entier
directement! *)
        pt_element = ^t_element;  (* le type t_element est déclaré plus tard *)
        ...
        t_element = record
                       nombre : integer;
                       element_suivant : pt_element; (* ici le type pointeur doit *)
                                                     (* déjà exister *)
                    end;
```

**Remarque**

Le nom d'un type pointeur commence souvent par ***pt_*** ce qui évite de préfixer par ***t_*** .

La seule **constante** possible est prédéfinie: **nil.** Elle correspond à une valeur nulle et peut être affectée à n'importe quelle variable pointeur. Elle permet donc d'initialiser un pointeur. Sur l'exemple précédent (cf. 9.2), elle pourrait être utilisée pour initialiser le champ *ville_reliée* de la dernière ville (Lausanne dans notre situation).

Les **opérations** sur les pointeurs (en plus de l'affectation et du passage en paramètre) sont réalisées par des procédures prédéfinies (cf. 9.3.3). Il est aussi possible d'accéder à la variable dynamique désignée par un pointeur de la manière suivante:

**var** pointeur_courant : pt_element;

Alors *pointeur_courant^* représente la variable dynamique (de type *t_element* ) pointée par le pointeur *pointeur_courant*.

***Cette variable dynamique s'utilise comme n'importe quelle variable du même type.***

Les **expressions** d'un type pointeur sont réduites à **nil**, aux variables et appels de fonction à résultat d'un type pointeur.

Il n'est pas possible de lire ou d'écrire (directement) la valeur d'un pointeur.

**Remarque:**

L'instruction **with** peut être utilisée avec une variable de type "pointeur sur un enregistrement":

```
var pointeur_courant : pt_element;
...
  with pointeur_courant^ do
        nombre := 10; (* nombre est un champ (entier) de pointeur_courant^ *)
  ...
  end;
```

[ Retour à la table des matières ]

### 9.3.2 **Affectation**

L'affectation s'effectue de manière habituelle entre une variable et une expression du **même type pointeur:**

**Exemple:**

```
var  pointeur_courant : pt_element;   (* cf. 9.3.1 *)
      element : t_element;             (* cf. 9.3.1 *)
...
      element.element_suivant := pointeur_courant;
      pointeur_courant := nil;
```

[ Retour à la table des matières ]

### 9.3.3 **Procédures et fonctions prédéfinies**

Il existe deux procédures prédéfinies:

- new ( variable_d_un_type_pointeur );

  Cette procédure crée une variable dynamique pointée par le pointeur *variable_d_un_type_pointeur*.

Cette variable dynamique est évidemment **non initialisée**.

- dispose ( variable_d_un_type_pointeur );

Cette procédure élimine la variable pointée par *variable_d_un_type_pointeur*.
**ATTENTION:** la valeur de *variable_d_un_type_pointeur* est alors indéfinie.

Il existe la fonction prédéfinie:

- ord ( variable_d_un_type_pointeur )

Elle donne la valeur (nombre entier) du pointeur *variable_d_un_type_pointeur* , c'est-à-dire l'adresse de la variable dynamique pointée par le pointeur *variable_d_un_type_pointeur*.

**Exemple:**

```
var pointeur : pt_element; (* cf. 9.3.1 *)
...
   new ( pointeur );         (* création d'une variable de type *)
                             (* element *)
   pointeur^.nombre := 0;    (* initialisation de cette variable *)
   pointeur^.element_suivant := nil;
   ...
   dispose ( pointeur );     (* élimination de cette variable *)
   pointeur := nil;          (* bonne habitude à prendre! *)
```

**Remarque**:

Si le type de la variable dynamique est un type enregistrement à variante la valeur de chaque variante doit être spécifiée à l'appel de *new* et *dispose* (cf. 12.1.3).

[ Retour à la table des matières ]

### 9.3.4 **Remarques importantes**

1. Une variable de type pointeur doit souvent être initialisée à **nil**.

2. Une fonction peut rendre un résultat d'un type pointeur (cf. 5.8).

3. Soit **var** pointeur : pt_element; (* cf. 9.3.1 *)

   Il ne faut pas confondre *pointeur* la variable pointeur de type *pt_element*
   et *pointeur^* la variable dynamique (de type *t_element* ) pointée
   par *pointeur*

4. Lorsque une variable dynamique devient inutile il faut l'éliminer avec *dispose*, puis
   réinitialiser le pointeur (qui pointait sur cette variable) à **nil.**

[ Retour à la table des matières ]

### 9.3.5 **Gestion des variables dynamiques**

Que se passe-t-il dans la mémoire de l'ordinateur lors de l'utilisation de variables dynamiques? La réponse est la suivante:

- lors de l'exécution de new ( *variable_d_un_type_pointeur* ) la place mémoire nécessaire à l'implantation de la variable pointée par *variable_d_un_type_pointeur* est **réservée pour cet usage**.

- lors de l'exécution de dispose ( *variable_d_un_type_pointeur* ); la place mémoire utilisée pour la variable pointée par *variable_d_un_type_pointeur* est **libérée.**

Il faut réaliser que la seule manière de libérer une place mémoire réservée pour de telles variables est d'utiliser dispose.

Que fait alors le morceau de programme suivant:

```
var pointeur    : pt_element; (* cf. 9.3.1 *)
      nb        : integer;
...

repeat
     readln ( nb );                    (* nombre donné par l'utilisateur *)
     new ( pointeur );
     pointeur^.nombre := nb;           (* insertion de ce nombre dans un champ *)
                                       (* d'une variable dynamique *)
     pointeur^.element_suivant := nil; (* pas de liens avec une autre variable *)
                                       (* dynamique *)
until nb = 0;
```

Chaque exécution du corps de la boucle crée une variable pointée par *pointeur* et contenant le nombre lu. Cela signifie aussi que les anciennes variables pointées par *pointeur* sont irrémédiablement perdues en mémoire! La place utilisée est gaspillée car désormais inutilisable.

Cette boucle peut donc remplir complètement la mémoire de zones inaccessibles si elle s'exécute suffisamment! On a donc la

***règle no 1:*** ne jamais rendre inaccessible des variables dynamiques sous peine de remplir la mémoire et de provoquer (éventuellement) un "crash" du système.

Que se passe-t-il si, pour respecter scrupuleusement la règle no 1, la procédure *dispose* est appliquée deux fois à une variable dynamique? La réponse est donnée sous la forme de la

***règle no 2:*** ne jamais exécuter *dispose* avec un pointeur de valeur indéfinie ou égale à **nil** sous peine de provoquer un arrêt brutal du programme.

On peut condenser ces deux règles en une seule qui paraît alors évidente:

***règle no 3 :*** toute variable dynamique créée par un appel à *new* doit être détruite une et une seule fois par un appel à *dispose* dès qu'elle devient inutile pour la suite du programme.

**Remarque**:

La fin de l'exécution d'un programme élimine toutes les variables dynamiques qu'il a créées. Cela signifie que la mémoire ainsi libérée est à nouveau disponible.

[ Retour à la table des matières ]

## 9.4 **Exemple complet**

### 9.4.1 **Enoncé du problème**

Ecrire un programme qui lit des nombres donnés par l'utilisateur et les enregistre dans l'ordre croissant. Lorsque tous les nombres ont été lus ils doivent être écrits dans cet ordre sur l'écran du terminal.

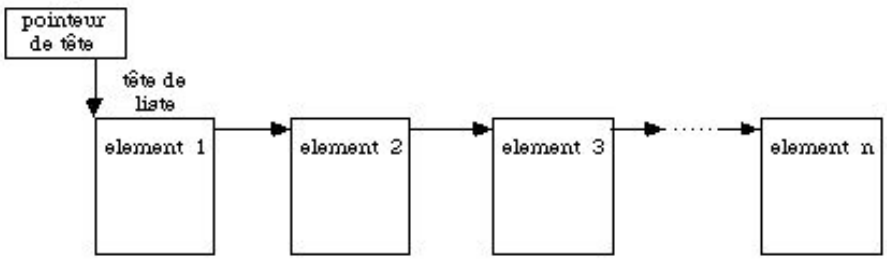
La résolution de ce problème n'est pas agréable si un tableau est utilisé (comment dimensionner le tableau? Comment insérer un nombre entre deux autres accolés?). C'est un cas typique d'utilisation des pointeurs.

[ Retour à la table des matières ]

### 9.4.2 **Décomposition du problème**

On va utiliser notre méthode de décomposition par raffinements successifs (cf. 2.2) afin de réduire notre problème en une succession de petits problèmes plus faciles à résoudre.

```
Premier raffinement: fixer les grandes lignes de l'algorithme.
    a) initialisations des objets utilisés
    b) répéter
         c) lire le nombre
         d) l'enregistrer au bon endroit dans la suite ordonnée des nombres déjà lus
       jusqu'à ce que tous les nombres aient été lus
    e) écrire tous les nombres dans l'ordre croissant
Deuxième raffinement: on va utiliser une liste pour représenter la suite ordonnée des
                      nombres. Une liste est composée d'éléments dont chacun
                      connaît l'élément suivant. Le premier élément est appelé tête
de
                   liste et est repéré par le pointeur de tête. Schématiquement:
```




```
L'algorithme devient:
    a) initialisations des objets utilisés
    b) répéter
         c) lire le nombre
        d1) créer un élément qui contiendra le nombre lu
        d2) chercher dans la liste la place où cet élément doit être introduit
        d3) l'introduire
       jusqu'à ce que tous les nombres aient été lus
    e) parcourir la liste en écrivant chaque nombre rencontré
```

```
Troisième raffinement: introduire dans l'algorithme l'utilisation des pointeurs.
                       Ceraffinement provoque la fusion des points d2 et d3
ci-dessus!
    a) initialisations des objets utilisés

    b) répéter
            c) lire le nombre

           d1) créer l'élément au moyen de new
           d2) si la liste est vide alors
                     introduire cet élément comme premier élément de la liste
               sinon  si le nombre est plus petit que celui de la tête de liste alors
                             introduire l'élément comme tête de liste
                      sinon  debut
                                 appelons courant le deuxième élément de la liste
                                 tant que l'insertion n'a pas été faite et
                                          l'élément courant existe faire
                                 debut
                                   si le nombre est plus petit que celui de courant
    alors
                                       introduire l'élément avant courant
                                   sinon courant devient l'élément suivant
                                 fin
                                 si l'insertion n'a pas encore été faite alors
                                        insérer l'élément à la fin de la liste
                             fin
       jusqu'à ce que tous les nombres aient été lus

    e) soit courant le premier élément de la liste
       tant que courant existe faire
       debut
           e1) écrire le nombre contenu dans courant
           e2) courant devient l'élément suivant
           e3) éliminer l'élément dont on a écrit le nombre
       fin
```

[ Retour à la table des matières ]

### 9.4.3 **Définition des types et des procédures**

L'algorithme ainsi construit nous permet de préciser les types et procédures nécessaires pour son codage en Pascal.

Types définis:

```
    type pt_element = ^t_element; (* type des liens entre éléments *)
                                   (* de la liste à construire. *)
          t_element = record      (* type des éléments de la liste *)
                        nombre : real;
                        suivant : pt_element;
                      end;
```

Procédures définies:

```
      procedure    initialiser;
                    (* initialise toutes les variables globales du programme *)

      procedure    lire_et_creer ( var p_element : pt_element );
```

```
                (* lit un nombre donné par l'utilisateur et crée l'élément *)
                (* correspondant. le paramètre p_element est le pointeur *)
                (* sur l'élément créé *)

    procedure   inserer ( var pt_de_tete : pt_element ;
                                p_element : pt_element );
                (* insère l'élément pointé par p_element au bon endroit dans *)
                (* la liste repérée par pt_de_tete. *)

    procedure   parcourir_ecrire ( var pt_de_tete : pt_element );
                (* parcourt la liste repérée par pt_de_tete , écrit chaque *)
                (* nombre rencontré et détruit la liste devenue inutile *)
```

[ Retour à la table des matières ]

### 9.4.4 **Programme final**

Commençons par donner le code du programme principal, puis explicitons le code de chacune des procédures.

```
program liste_triee ( input, output );
(* Auteur: ... *)
(* But du programme: Demander à l'utilisateur une suite de nombres se *)
(*                   terminant par zero. Le programme triera cette suite par *)
(*                   ordre croissant et écrira les nombres dans cet ordre. *)
(*                                                          19.1.87 PBT *)

  const nombre_final = 0.0;             (* nombre terminant la suite donnée *)

  type pt_element = ^t_element;         (* type des liens entre éléments *)
                                         (* de la liste à construire *)

       t_element = record               (* type des éléments de la liste *)
                     nombre : real;
                     suivant : pt_element;
                   end;

  var pt_de_tete : pt_element;          (* pointeur sur la tête de la liste *)
      pt_courant : pt_element;           (* pointeur sur l'élément courant *)

(* Ici sont déclarées les 4 procédures utilisées. Seuls les en-têtes *)
(* sont présents pour rendre le programme principal compréhensible. *)

  procedure     initialiser;
               (* initialise toutes les variables globales du programme *)

  procedure     lire_et_creer ( var p_element : pt_element );
               (* lit un nombre donné par l'utilisateur et crée l'élément *)
               (* correspondant. Le paramètre p_element est le pointeur *)
               (* sur l'élément créé *)

  procedure     inserer ( var pt_de_tete : pt_element ;
                              p_element : pt_element );
               (* insère l'élément pointé par p_element au bon endroit dans *)
               (* la liste repérée par pt_de_tete. *)

  procedure     parcourir_et_ecrire ( var pt_de_tete : pt_element );
               (* parcourt la liste repérée par pt_de_tete , écrit chaque *)
               (* nombre rencontré et détruit la liste devenue inutile *)
```

```
begin (* liste_triee *)

   (* initialiser les variables du programme principal *)
   initialiser;

   (* message de présentation du programme, non explicité ici *)

   repeat     (* saisir la suite des nombres de l'utilisateur *)

      (* créer l'élément correspondant au nombre donné par l'utilisateur *)
      (* y compris le nombre final. *)
      lire_et_creer ( pt_courant );

      (* insérer l'élément courant dans la liste au bon endroit *)
      inserer ( pt_de_tete, pt_courant );

   until  pt_courant^.nombre = nombre_final;

   (* écrire le suite des nombres triés par ordre croissant, détruire la liste *)
   parcourir_et_ecrire ( pt_de_tete );

end.
```

Voici le code de chacune des procédures:

```
procedure initialiser;
(* initialise toutes les variables globales du programme *)

begin (* initialiser *)
    pt_de_tete := nil;         (* la liste est vide *)
    pt_courant := nil;         (* inutile mais par habitude...! *)
end;  (* initialiser *)


procedure  lire_et_creer ( var p_element : pt_element );
            (* lit un nombre donné par l'utilisateur et crée l'élément *)
            (* correspondant. Le paramètre p_element est le pointeur *)
            (* sur l'élément créé *)

begin (* lire_et_creer *)
   (* créer et initialiser l'élément correspondant *)
    new ( p_element );
    p_element^.suivant := nil;       (* ne pas oublier cela! *)

   (* demander le nombre suivant *)
    write ( 'Quel est le nombre suivant : ');
    readln ( p_element^.nombre );
end; (* lire_et_creer *)


procedure parcourir_et_ecrire ( var pt_de_tete : pt_element );
           (* parcourt la liste repérée par pt_de_tete , écrit chaque *)
           (* nombre rencontré et détruit la liste devenue inutile. Le *)
           (* pointeur pt_de_tete vaut nil à la sortie de cette procédure. *)
```

```
    var pt_inutile : pt_element;              (* élément que l'on élimine *)

begin (* parcourir_et_ecrire *)
    while pt_de_tete <> nil do              (* tous les éléments un à un *)
    begin
        (* écriture du nombre courant *)
        writeln ( 'Le nombre suivant est : ', pt_de_tete^.nombre : 8 );

        pt_inutile := pt_de_tete;          (* repérer l'élément devenu inutile *)
        pt_de_tete := pt_de_tete^.suivant; (* passer à l'élément suivant *)
        dispose ( pt_inutile );            (* éliminer l'élément inutile *)
    end;

end; (* parcourir_et_ecrire *)

procedure inserer ( var pt_de_tete : pt_element ;
                        p_element : pt_element );
          (* insère l'élément pointé par p_element au bon endroit dans *)
          (* la liste repérée par pt_de_tete. *)

       var pt_courant : pt_element;       (* pointeur sur l'élément courant *)
           pt_precedent : pt_element;     (* pointeur sur l'élément qui *)
                                          (* précède l'élément courant *)
           est_insere : boolean;          (* vrai lorsque l'élément a été *)
                                          (* inséré *)
begin (* inserer *)
   if pt_de_tete = nil then     (* liste vide, introduire le 1er élément *)
       pt_de_tete := p_element

    else if p_element^.nombre < pt_de_tete^.nombre then
          begin                          (* nouvelle tête de liste *)
              p_element^.suivant := pt_de_tete;
              pt_de_tete := p_element;
          end

          else begin (* inserer plus loin que la tête de liste *)
                     pt_courant := pt_de_tete^.suivant; (* 2ème élément *)
                     pt_precedent := pt_de_tete;        (* 1er élément *)
                     est_insere := false;

                while (pt_courant <> nil) and not est_insere do
                   begin
                      (* faut-il insérer avant l'élément pointé par pt_courant? *)
                       est_insere := p_element^.nombre < pt_courant^.nombre;

                    if est_insere then
                    begin  (* inserer avant l'élément pointé par pt_courant *)
                           p_element^.suivant := pt_courant;
                           pt_precedent^.suivant := p_element;
                    end
                    else begin   (* passer à l'élément suivant de la liste *)
                                 pt_precedent := pt_courant;
                                 pt_courant := pt_courant^.suivant;
                           end;
                 end; (* while *)

                 if not est_insere then  (* insertion à la fin de la liste *)
```

```
                        pt_precedent^.suivant := p_element;
                end;

    end; (* inserer *)
```

[ Retour à la table des matières ]

**CHAPITRE 12**

# COMPLEMENTS DE PASCAL

## 12.1 **Enregistrements à partie variante**

### 12.1.1 **Motivation**

Le type enregistrement est utilisé pour représenter des données formées d'éléments de types différents (cf. 6.5). Or il existe des cas où certaines composantes de ces données diffèrent en fonction de l'utilisation, par exemple:

- les fiches personnelles diffèrent selon le sexe de la personne (nom de jeune fille d'une femme mariée!)

- une fiche bibliographique est complétée par des renseignements sur l'emprunteur lorsque le livre est prêté

- les nombres complexes sont représentés avec parties réelle et imaginaire ou par module et argument.

Il est cependant naturel de représenter de telles données par des enregistrements. Pascal met donc à disposition un type enregistrement plus complet que celui vu jusqu'à présent: le type **enregistrement à (partie) variante**.

Notons cependant que le type enregistrement vu jusqu'ici (cf. 6.5) s'obtient lorsque la partie variante n'existe pas!

[ Retour à la table des matières ]

### 12.1.2 **Généralités**

Un type enregistrement à variante se déclare en spécifiant une partie variante comme **dernier** élément d'un enregistrement. Par exemple:

```
type t_annee_de_parution = 1500 .. 2099;

      t_fiche_biblio = record
                         auteur : t_nom; (* t_nom est un type chaîne de caractères
*)
                         titre : ...; (* type non précisé *)
                         annee_de_parution : t_annee_de_parution;
                         case prete : boolean of                (* partie variante
*)
                                  false : ( );
                                  true : ( nom_emprunteur : t_nom;
                                           date_emprunt : date; (* pour le type *)
                                           date_retour : date ); (* date , cf. 6.2
*)
                       end; (* fin de la partie variante et de l'enregistrement *)

       t_representation = ( polaire, cartesienne );

       t_complexe = record
                      case mode_utilisation : t_representation of

                        cartesienne : ( partie_reelle : real;
                                        partie_imaginaire : real );
                        polaire     : ( module : real;
```

```
                                        argument : real );
                    end;
```

La structure **case** permet ici de définir le **champ indicateur** et, pour chacune des valeurs mentionnées (constantes!) de définir la structure de cette composante de la partie variante.

La déclaration de variables s'effectue comme habituellement. C'est l'utilisation de telles variables qui va être nouvelle:

```
    var manuel_pascal : t_fiche_biblio;

begin
   manuel_pascal.auteur := 'N. Wirth';
   manuel_pascal.titre := ...;
   manuel_pascal.annee_de_parution := 1971;
   manuel_pascal.prete := true;        (* la composante existante est alors celle *)
                                       (* spécifiée lorsque le champ prete vaut true
*)
   manuel_pascal.nom_emprunteur := 'J. Dupont';
   ...
```

**Attention:** il ne devrait pas être possible d'accéder à des champs d'une composante non existante (mais certains compilateurs ne le vérifient pas!)

**Exemple:**

```
manuel_pascal.prete := false; (* la composante existante est alors vide! *)
manuel_pascal.nom_emprunteur := 'J. Dupont'; (* ERREUR *)
```

[ Retour à la table des matières ]

12.1.3 **Remarques**

1. Noter l'utilisation des parenthèses, la présence éventuelle d'une composante vide, l'absence possible de la partie fixe lors de la déclaration d'une partie variante (cf. 12.1.2 ).

2. Le type du champ indicateur doit être scalaire (cf. Annexe F).

3. Il est possible mais fortement déconseillé de remplacer le champ indicateur par le seul nom du type, par exemple:

```
type    t_representation = ( polaire, cartesienne );

        t_complexe = record
                        case representation of
                          cartesienne : ( partie_reelle : real;
                                          partie_imaginaire : real );
                          polaire     : ( module : real;
                                          argument : real );
                     end;
```

   Dans ce cas toutes les composantes de la partie variante sont actives donc utilisables!

4. Il est possible de déclarer une partie variante comme dernier élément de la dernière composante d'une partie variante!

5. L'utilisation des procédures standard new et dispose sur des pointeurs désignant un enregistrement à variante est particulière:

   Il faut spécifier après la variable pointeur la valeur de tous les champs indicateurs de l'enregistrement pointé. De plus ces

champs indicateurs ne pourront pas changer de valeur; celles-ci doivent d'ailleurs être explicitement affectées à ces champs après exécution de new!!!

**Exemple:**

```
...
new ( pt, false );          (* si pt est de type ^t_fiche_biblio  (cf. 12.1.2) *)
pt^.prete := false;         (* affectation explicite obligatoire *)
...
dispose ( pt, false );      (* même valeur que pour new *)
```

[ Retour à la table des matières ]

## 12.2 **Sauts inconditionnels**

Comme dans n'importe quel langage l'instruction **goto** (mot réservé) existe aussi en Pascal. Son effet est de provoquer un saut dans l'exécution du programme.

**Exemple:**

```
program horrible (input, output );
(* Auteur : M. Frankenstein *)
(* ... *)

   label 33, 134, 8888; (* 134 et 8888 non utilisés *)

   var   i : integer;   (* ... *)

begin (* horrible *)
   ...
33 :  i := 0;     (* forme une boucle entre cette instruction d'affectation ... *)
      ...
   goto 33;      (* ... et cette instruction de saut. *)
...
end.
```

L'exécution de ce programme s'effectuera comme habituellement jusqu'à l'instruction **goto** y compris puis continuera à l'instruction qui suit le nombre 33 qui est appelé **étiquette**. On a donc un saut à travers le programme.

**Règles:**

- Toute étiquette est un nombre entier compris entre 1 et 9999. Elle doit être déclarée dans une zone de déclaration commençant par le mot réservé **label**. Cette zone précède celle de déclaration des constantes.

- Une étiquette se place devant n'importe quelle instruction, séparée de celle-ci par le symbole : . Cette instruction est celle qui sera exécutée en premier après un saut à cette étiquette.

- La portée d'une étiquette est définie comme pour n'importe quel autre objet déclaré ( cf. 5.6 ) .

- Il n'est pas possible de sauter n'importe où depuis n'importe que lieu. Certaines restrictions ( dépendant des compilateurs ) existent !

**AVERTISSEMENT**

L'utilisation des étiquettes et goto doit être réservée à des cas **exceptionnels** où de tels sauts sont utiles. En effet la lisibilité des programmes est fortement diminuée et leur structure détruite par la présence de sauts. De plus ceux-ci peuvent conduire à un programme dont l'effet est imprévisible! Un programme comme ci-dessous est à bannir absolument:

```
program affreux_et_a_effet_imprevisible ( input, output );
(* Auteur: Mr. Hyde *)
(* ... *)

   label 1;

   const max = 10;       (* ... *)

   var   i : integer;    (* variable de boucle *)

begin (* affreux_et_a_effet_imprevisible *)

   ...
   goto 1;
   ...
   for i := 1 to max do
   begin
       ...
    1: ...    (* quelle est la valeur de i ? *)
       ...
   end;
  ...
end.
```

Personne n'est capable de dire quel est l'effet exact de ce programme!

[ Retour à la table des matières ]

## 12.3 **Passage de procédures et fonctions en paramètre**

### 12.3.1 **Exemple introductif**

Soit le problème suivant:

> Ecrire un programme donnant l'ordonnée d'une fonction pour une abscisse donnée. Les fonctions disponibles et l'abscisse sont choisies par l'utilisateur.

Les notions de Pascal connues jusqu'à présent sont suffisantes pour résoudre ce problème mais le programme obtenu serait relativement complexe si la liste des fonctions disponibles est grande. En effet tous les calculs qui dépendent des fonctions doivent être répétés autant de fois qu'il y a de fonctions!

Pour simplifier cela Pascal fournit le passage de procédures et fonctions en paramètre. Pour notre problème il va donc être possible d'écrire une fonction de calcul évaluant l'ordonnée (en un point) de n'importe quelle fonction, celle-ci étant passée en paramètre de notre fonction de calcul!

La fonction de calcul s'écrit alors comme suit:

```
function ordonnee ( abscisse : real;
                    function f ( x : real ) : real ) : real;
(* calcule la valeur de la fonction f au point abscisse *)

begin (* ordonnee *)
```

```
   ordonnee := f ( abscisse );      (* calcul de l'ordonnée *)

end; (* ordonnee *)
```

Cette fonction peut alors s'appeler de la manière suivante:

- ordonnee ( 10.0, sqrt ); (* calcule la racine carrée de 10.0 *)

- ordonnee ( 3.14 / 4.0, sin ); (* calcule le sinus de pi sur 4 *)

- ordonnee ( -2.1, polynome ); (* calcule la valeur de polynome ( *-2.1* ) si *)
  (* l'en-tête de *polynome* comporte un et un *)
  (* seul paramètre de type *real* (voir ci-dessous) *)

[ Retour à la table des matières ]

### 12.3.2 **Généralités**

Toute procédure ou fonction peut être passée en paramètre. Pour ceci il faut que son en-tête ainsi que la déclaration de paramètre soient **identiques** exception faite

- des noms des paramètres formels
- du nom du paramètre (procédure ou fonction)

**Exemples:**

```
procedure cube ( nombre : integer;
                 var resultat : integer );
begin (* cube *)
   ...
end; (* cube *)
```

peut être passée comme paramètre effectif de la procédure

```
   procedure calcul ( procedure proc ( n : integer;
                                       var r : integer ) );
```

ou comme paramètre de la fonction

```
   function evaluation ( procedure test ( i : integer;
                                          var r : integer ) ) : real;
```

mais **pas** comme paramètre de la procédure

```
   procedure faux ( procedure proc ( var n : integer;
                                     var r : integer ) );
```

puisque le mode de passage du premier paramètre diffère.

**Remarque**

Certains Pascal (comme Pascal Macintosh!) ne permettent pas de passer les procédures et fonctions **prédéfinies** comme paramètres.

[ Retour à la table des matières ]

### 12.3.3 **Exemple complet**

Reprenons notre problème introductif (cf. 12.3.1). Voici le programme demandé fonctionnant sur Macintosh:

```
program exemple ( input, output );
(* Auteur : ...                                                    *)
(* Ce programme calcule l'ordonnee d'une fonction en un point. La fonction *)
(* et le point sont choisis par l'utilisateur.                     *)
(* Attention: aucun test de débordement n'est fait                 *)

   const  nb_fonctions = 3;   (* nombre de fonctions à choix *)

   var    point : real;       (* point choisi pour le calcul *)
          choix : integer;    (* fonction choisie pour le calcul *)
          valeur : real;      (* valeur de la fonction au point choisi *)

   function polynome ( abscisse : real ) : real;
   (* calcule la valeur du polynôme x2 + x - 3 *)

   begin (* polynome *)
      polynome := sqr ( abscisse ) + abscisse - 3.0;
   end; (* polynome *)

   function sinus ( abscisse : real ) : real;
   (* puisque Pascal Macintosh ne permet pas de passer sin directement... *)

   begin (* sinus *)
      sinus := sin ( abscisse );
   end; (* sinus *)

   function cosinus ( abscisse : real ) : real;
   (* puisque Pascal Macintosh ne permet pas de passer cos directement... *)

   begin (* cosinus *)
      cosinus := cos ( abscisse );
   end; (* cosinus *)

   function ordonnee ( abscisse : real;
                       function f ( x : real ) : real ) : real;
   (* calcule la valeur de la fonction f au point abscisse *)

   begin (* ordonnee *)
      ordonnee := f ( abscisse ); (* calcul de l'ordonnée *)
   end; (* ordonnee *)

begin (* exemple *)

   (* présentation du programme et choix de la fonction *)
   writeln;
   writeln ('Je suis capable de vous calculer la valeur d''une fonction en un ');
   writeln (' point donne. Veuillez choisir la fonction dans la liste suivante:');
   writeln;
   writeln(' 1 sinus');
   writeln(' 2 cosinus');
   writeln(' 3 polynome x2 + x - 3');
   writeln;
   write ( 'Votre choix: ');
   readln ( choix );
```

```
      (* choix correct? *)
      if not ( choix in [1..nb_fonctions] ) then
         writeln ( 'Erreur de choix')
      else begin

              (* détermination du point *)
              write ( 'Valeur du point: ');
              readln ( point );

              (* calcul du point *)
              case choix of
                  1 : valeur := ordonnee ( point, sinus );
                  2 : valeur := ordonnee ( point, cosinus );
                  3 : valeur := ordonnee ( point, polynome );
             end;

             (* écriture de la valeur de la fonction au point choisi *)
              writeln ( 'La fonction vaut ', valeur : 8 : 3, ' au point ', point : 8 :
   3 );

          end;

   end.
```

[ Retour à la table des matières ]

## 12.4 **Les structures empaquetées**

### 12.4.1 **Généralités**

Pour comprendre ce paragraphe, il faut savoir comment sont implantées en mémoire les structures de données telles que entiers, réels, tableaux, enregistrements... Cette implantation est réalisée dans la mémoire de l'ordinateur au moyen des ses éléments appelés **mots**. Un mot mémoire est formé d'un certain nombre de bits, en général 8, 16, 32, ..., 64, ... bits.

Cela signifie que chaque constante ou variable va occuper un ou plusieurs mots, par exemple

- un entier occupe (en général) un mot
- un réel occupe (en général) deux mots

Mais

- un caractère occupe (en général) les 8 premiers bits d'un mot
- un booléan nécessite même qu'un seul bit

Ceci n'est pas grave lorsqu'un programme n'utilise que quelques objets de ces types. Or la place mémoire non utilisée peut devenir rapidement importante lorsque des enregistrements ou de tableaux constuits sur ces types sont déclarés! Il peut alors être précieux d'économiser la place mémoire en "tassant" les valeurs à l'intérieur des mots. On dit dans ce cas que les structures de données sont **empaquetées**.

Une structure empaquetée se déclare au moyen du mot réservé **packed**. Seuls les tableaux, enregistrements, ensembles et fichiers peuvent être déclarés comme empaquetés.

**Exemples:**

```
   const max = 100;
   type  t_mot = packed array [ 1 .. max ] of char;

         t_date = packed record
```

```
    jour : t_jours_mois;
    mois : t_mois_de_l_annee; (* cf. 7.1.2 *)
    annee : integer;
end;
```

**Remarque importante**

Cette technique ne change ni les résultats fournis par le programme ni les bonnes habitudes du programmeur! Elle conduira simplement à écrire des programmes légèrement moins lisibles, un peu plus lents mais occupant moins de place en mémoire.

[ Retour à la table des matières ]

### 12.4.2 **Avantages des structures empaquetées**

L'avantage le plus important est le gain de place obtenu grâce au tassement des valeurs. Mais certains autres avantages sont obtenus de manière indirecte. Citons

- l'utilisation de chaînes de caractères souvent déclarées comme suit:

  **packed array** [ intervalle ] of char

- l'utilisation des opérateurs de comparaison pour de telles chaînes

[ Retour à la table des matières ]

### 12.4.3 **Inconvénients des structures empaquetées**

Le principal inconvénient est l'accroissement du temps d'accès à une valeur empaquetée. En effet des instructions-machine supplémentaires sont générées par le compilateur pour lire ou modifier une telle valeur. Ceci prend un peu de temps et occupe également de la place en mémoire! Il faut donc n'empaqueter que les grandes structures de données.

Un autre problème se pose avec le passage de paramètre: il n'est en général pas possible de passer en paramètre un élément d'une telle structure. Il faut d'abord "désempaqueter"
la structure.

[ Retour à la table des matières ]

### 12.4.4 **Les procédures prédéfinies *pack* et *unpack***

Deux procédures prédéfinies permettent d'empaqueter et de "désempaqueter" un tableau; ce sont *pack* et *unpack*:

Soient les déclarations suivantes:

```
const   m1 = ...;
        m2 = ...;
        n1 = ...;
        n2 = ...;

type    T = ...;
        t_tab_normal = array [ m1 .. n1 ] of T;
        t_tab_empaquete = packed array [ m2 .. n2 ] of T;

var    tab_normal : t_tab_normal;
       tab_empaquete : t_tab_empaquete;
```

L'appel à la procédure *pack* va empaqueter le tableau *tab_normal* dans le tableau *tab_empaquete* . On a que

```
pack ( tab_normal, indice, tab_empaquete ); est équivalent à l'instruction

for j := m2 to n2 do
```

```
tab_empaquete [ j ] := tab_normal [ j - m2 + indice ]
```

Ceci signifie "remplir le tableau *tab_empaquete* par les éléments *tab_normal [ indice ], tab_normal [ succ (indice) ]*..."

L'appel à la procédure *unpack* va "désempaqueter" le tableau tab_empaquete dans le tableau *tab_normal* . On a que

```
unpack ( tab_empaquete, tab_normal, indice );
```

est équivalent à l'instruction

```
for j := m2 to n2 do
   tab_normal [ j - m2 + indice ] := tab_empaquete [ j ]
```

Ceci signifie "remplir le tableau *tab_normal* dès l'élément d'indice indice par le tableau *tab_empaquete* .

**Remarque**

Pour que ceci fonctionne il faut que l'intervalle *m2* .. *n2* comporte **moins ou autant** de valeurs que l'intervalle *m1* .. *n1*.

[ Retour à la table des matières ]

**CHAPITRE 6**

# LES TABLEAUX ET LES ENREGISTREMENTS

## 6.1 **Motivation**

Notre connaissance des types et des instructions en Pascal est relativement limitée jusqu'à présent. Nous avons vu les 4 types de base (*integer, char, boolean, real*) et 6 instructions (**if, while, for**, affectation, instruction composée, appel de procédure).

Nous allons approfondir nos connaissances dans deux directions, à savoir:

- examiner attentivement la notion de **type** qui va permettre une représentation plus agréable mais plus complexe de l'information (données) traitée dans les programmes. Les types que nous allons étudier dans ce chapitre sont les types **tableau** et **enregistrement.**

- manipuler les **instructions** complémentaires à celles déjà connues permettant le traitement de l'information manipulée dans les programmes. Dans ce chapitre nous verrons les instructions **repeat** et **with.**

**Remarque**

Les chapitres 7, 8, 9 et 10 présentent tous les autres types de Pascal à savoir les types **énumérés**, **intervalle, ensemble, chaînes de caractères, pointeurs** et **fichiers** ainsi que la dernière instruction **(case)**.

Ceci va nous permettre de réaliser des **programmes importants** non seulement du fait de leur corps souvent volumineux mais aussi (surtout) à cause de leur partie déclarative contenant des données complexes.

Les données ainsi que la manière de les représenter forment ce que l'on appelle les **structures d'information** ou **structures de données** des programmes.

[ Retour à la table des matières ]

## 6.2 **Notion de type**

Un **type** est formé **d'un groupe de valeurs et d'opérations** possibles sur ces valeurs.

**Exemples:**

1. Le type integer (cf. 3.2) est formé d'un groupe de valeurs entières dont la grandeur dépend de l'ordinateur utilisé et
   - des opérations + - * **div mod**
   - des opérations de comparaison

   Lorsque l'on parle de "valeur de type *integer*" cela veut dire "une valeur appartenant aux nombres entiers représentables et avec laquelle on peut additionner, soustraire, multiplier, diviser, prendre le reste de la division ..."

2. Le type char (cf. 4.5) est formé d'un groupe de 128 (ou 256) caractères et des opérations de comparaison.

Pour pouvoir utiliser un type il faut d'abord le déclarer. Cette déclaration doit survenir dans une zone (de la partie déclarative) située entre les déclarations de constantes et de variables. Cette zone commence par le mot réservé **type** et comprend la déclaration de tous les types utilisés dans le programme, la procédure ou la fonction concerné.

Chaque catégorie de type possède son style de déclaration. Seul le début de la déclaration d'un type est commun à tous les types:

***nom_du_type*** =

**Exemples:**

```
type jours_de_la_semaine = ( lundi, mardi, mercredi, jeudi,
                             vendredi, samedi,dimanche ); (* cf. 7.1 *)

     mois_de_l_annee =     ( janvier, fevrier, mars, avril, mai,
                             juin, juillet, aout,septembre,
                             octobre, novembre, decembre );
                                                  (* cf. 7.1 *)

     jours_de_travail = lundi..vendredi;          (* cf. 7.3 *)

     jours_du_mois = 1..31;                       (* cf. 7.3 *)

     vecteur = array [1..10] of real;             (* cf. 6.3 *)

     date = record                                (* cf. 6.5 *)
            jour  : jours_du_mois;
            mois  : mois_de_l_annee;
            annee : integer;
        end;

     teinte = set of couleurs_arc_en_ciel; (* cf. 8.1 *)

     nom = string [ 30 ];                         (* cf. 8.2, syntaxe Pascal Macintosh *)

     lien_ville = ^ville;                         (* cf. 9.3.1 *)

     fich_entiers = file of integer;              (* cf. 10.3.1 *)
```

La notion de type va donc permettre de catégoriser les données traitées ce qui implique une **sécurité accrue** pour le programmeur (moins d'erreurs de programmation), une représentation et un traitement plus agréable et plus "naturel" de ces mêmes données.

Il n'y a en effet rien de plus difficile et horrible à comprendre que des programmes où par exemple le nombre 2 possède plusieurs significations différentes: le deuxième jour de la semaine, la deuxième coordonnée du point (x,y), le deuxième élément de l'ensemble truc ...

Le programmeur va pouvoir **construire lui-même ses types** et ses structures d'information en utilisant les types prédéfinis et les constructeurs de types que nous allons voir dans les paragraphes suivants.

**Remarques:**

1. L'ordre des déclarations dans une partie déclarative devient:

- premièrement les constantes
- deuxièmement les types
- troisièmement les variables

2. Par convention, on peut faire précéder le nom du type par le préfixe ***t_*** qui permet de préciser que cet identificateur est le nom d'un type (et non d'une variable par exemple).

[ Retour à la table des matières ]

## 6.3 **Les types tableau**

### 6.3.1 **Motivation**

Dans de nombreuses situations le programmeur ressent le besoin de manipuler des données plus complexes que de simples valeurs. Par exemple:

- calculs vectoriel et matriciel
- gestion de structures telles que piles, listes ...
- gestion des tampons d'entrées-sorties dans les systèmes d'exploitation
...

Ces données ont la particularité d'être composées de plusieurs valeurs **de même type**. En effet un vecteur est un n-tuple de valeurs réelles, un tampon d'entrée-sortie contient plusieurs dizaines de caractères... Le lecteur réalise alors qu'une telle donnée est complètement définie lorsque

a) le **nombre** de valeurs composant cette donnée est défini
b) le **type** de ces valeurs est défini

Un **tableau** est une structure d'information (cf. 6.1) qui permet de représenter de telles données.

[ Retour à la table des matières ]

### 6.3.2 **Généralités**

Un tableau (**array**) est formé d'un ou de plusieurs **éléments** (ou **composantes**) tous de même type. Un **indice** est associé au tableau et permet de **numéroter** et de **distinguer** les éléments.

**Exemples:**

```
- const max = 10;                    (* cf.6.3.5 *)

  array [ 1..max ] of integer        représente un tableau de 10 éléments
                                      de type integer (tableau d'entiers).
                                      L'indice permet de numéroter et de
                                      distinguer les éléments du tableau.

- const nb_max_notes = 4;

  array [ 1..nb_max_notes ] of real  représente un tableau de réels.
                                      L'indice permet de distinguer
                                      les 4 éléments.
```

Les types tableau se déclarent ainsi:

```
  const  max = 10;                   (* cf. 6.3.5 *)
         nb_max_notes = 4;
         plus_petite_unite = -9;
         plus_grande_unite = 9;
```

```
    type   t_table = array [ 1..max ] of integer;
           t_notes = array [ 1..nb_max_notes] of real;
           t_unites = array [ plus_petite_unite .. plus_grande_unite ] of boolean;

et des variables de ces types:

    var   premiere_table,
          seconde_table : t_table;

           notes : t_notes;

           unites : t_unites;
```

Les types tableau font partie des types structurés (cf. annexe F). Concernant les types des indices et des éléments il faut savoir que

- le type des indices peut être *integer, char, boolean*, un type énuméré (cf. 7.1) ou un type intervalle (cf. 7.3) construit sur un de ces types;
- le type des éléments peut être **n'importe lequel** (sauf un type fichier, cf. 10.3);
- les bornes de l'intervalle doivent être des constantes.

Mais:

- Il n'existe pas de **constante** d'un type tableau.

- Il n'existe pas d'**opération** (autre que l'affectation et le passage comme paramètre) sur les tableaux.

- Les seules **expressions** d'un type tableau sont les variables de ce type.

- Il n'y a pas de **fonction** (même prédéfinie) à résultat d'un type tableau.

- Il n'y a pas d'**entrées-sorties** sur les tableaux.

Ces limitations rendent les tableaux inutilisables tels quels. En fait la majorité des cas d'utilisation des tableaux réside dans l'utilisation de leurs éléments.

[ Retour à la table des matières ]

### 6.3.3 **Accès aux éléments d'un tableau**

Comment accéder aux éléments d'un tableau? Un tel élément est défini par le nom du tableau ainsi que par une valeur de l'indice. Par exemple:

- *premiere_table [ 1 ]* est l'élément d'indice *1* du tableau *premiere_table.*

- si *index* est une variable entière de valeur 3, *premiere_table [ index ]* est l'élément d'indice 3 du tableau *premiere_table*.

- *seconde_table [ index * 3 - 5 ]* est défini pour des valeurs d'*index* de 2, 3, 4 et 5, inexistant sinon!

L'accès à un élément se fait donc en nommant le tableau et en spécifiant entre [ ] une expression donnant la valeur d'indice. Le résultat de cette construction est un élément de tableau du type spécifié à la déclaration du tableau, utilisable de manière

**absolument identique à une variable (simple) de ce type.**

Donc partout où une variable d'un type (simple) donné peut être utilisée, un élément de tableau de ce type peut l'être également.

**Remarque:**

Une tentative d'accéder à un élément de tableau inexistant (indice hors de l'ensemble de définition) provoque une erreur à l'exécution du programme.

[ Retour à la table des matières ]

### 6.3.4 **Affectation**

L'affectation se fait de manière habituelle entre **tableaux strictement de même type**. Par exemple:

```
premiere_table := seconde_table; (* cf. 6.3.2 *)
```

[ Retour à la table des matières ]

### 6.3.5 **Remarques**

1. La déclaration (correcte) suivante:

```
type t_table = array [ 1..10 ] of integer;
```

devrait toujours être remplacée par la suite de déclarations suivante:

```
const  max_elements = 10; (* nombre maximum d'éléments de la table *)

type    t_indice_table = 1..max_elements; (* intervalle de variation des *)
                                          (* indices, cf. 7.3 *)

        t_table = array [ indice_table ] of integer; (*...*)
```

ou au minimum par:

```
const  max_elements = 10; (* nombre maximum d'éléments de la table *)

type   t_table = array [ 1..max_elements ] of integer; (*...*)
```

2. Il est possible mais fortement déconseillé de déclarer des variables de la manière suivante:

```
var    premiere_table : array [ 1..max_elements ] of integer; (* cf. 6.3.2 *)
```

[ Retour à la table des matières ]

### 6.3.6 **Tableaux multidimensionnels**

Les tableaux vus jusqu'à présent sont appelés **unidimensionnels** car ils permettent de représenter un objet à une dimension, comme par exemple un vecteur.

Or le type des éléments d'un tableau peut être lui-même un tableau (cf. 6.3.2) et ainsi de suite. Il est donc par exemple possible de représenter des matrices grâce à des tableaux à 2 dimensions.

Un type tableau à deux dimensions se déclare par exemple ainsi:

```
const max_lignes = 20;  (* nb de lignes de la matrice *)
```

```
      max_colonnes = 5; (* nb de colonnes de la matrice *)

type  t_ligne = array [1..max_colonnes] of real;
      t_matrice_2_dim = array [1..max_lignes] of t_ligne;

var   matrice : t_matrice_2_dim; (* tableau à deux dimensions *)
```

L'accès aux éléments de ce tableau s'effectue de la manière suivante:

- matrice [1] représente l'élément d'indice 1, c'est-à-dire un tableau à une dimension (une ligne)!

- matrice [1][3] représente l'élément d'indice 3 de l'élément (ligne) d'indice 1, c'est-à-dire un nombre réel

Il existe des abréviations pour les tableaux multidimensionnels. Pour l'exemple ci-dessus une manière équivalente d'écrire les choses est:

```
type t_matrice_2_dim = array [1..max_lignes, 1..max_colonnes] of real;

var matrice : t_matrice_2_dim;
```

ainsi que l'accès aux composantes:

```
matrice [1, 3]    forme abrégée de matrice [1][3]
```

La notation abrégée a cependant comme conséquence que l'élément *matrice [1]* n'est plus défini donc inutilisable!

Tout ceci se généralise à n dimensions de façon identique.

[ Retour à la table des matières ]

### 6.4 **Itération: la boucle repeat**

L'instruction **repeat,** à l'instar de **while** (cf. 4.4), permet la répétition d'un groupe d'instructions un nombre de fois non connu à l'avance. Cette répétition s'effectue jusqu'à ce qu'une condition (expression booléenne) soit vraie.

**Exemple** (comparer avec **while**):

```
   const nombre_final = 0;  (* permet de terminer le programme lorsque *)
                            (* l'utilisateur le donne *)
   var nombre : integer;    (* nombre donné par l'utilisateur *)

begin (* ... *)

   (* initialisation de nombre pas nécessaire ici! *)

   repeat (* traiter les nombres de l'utilisateur *)
      write ( 'Donnez la valeur suivante: ' );
      readln ( nombre );
      if nombre <> nombre_final then
         ... (* traitement du nombre *)
   until nombre = nombre_final;
   ... (* instructions suivantes *)
```

Les instructions comprises entre les mots réservés **repea**t et **until** sont répétées jusqu'à ce que la condition *nombre = nombre_final* soit vraie.

La forme générale est:

```
repeat
    instruction;
    instruction;
    ...
until expression_booléenne;
```

**Remarques:**

1. Il faut s'assurer que la condition devient vraie après un nombre fini d'itérations pour éviter une boucle infinie.

2. Les instructions comprises entre **repeat** et **until** n'ont pas besoin d'être englobées par une instruction composée puisqu'elles le sont par **repea**t et **until** eux-mêmes.

3. Les instructions comprises entre **repeat** et **until** sont **exécutées au moins une fois**, quelle que soit la valeur de la condition. C'est la principale différence avec **while.**

4. La sémantique de **while** veut que la boucle soit exécutée **tant qu**'une condition est vraie, alors que celle de **repeat** impose que la boucle soit exécutée **jusqu'à ce qu**'une condition soit vraie (cf. exemples de 4.4 et 6.4)

[ Retour à la table des matières ]

## 6.5 **Les types enregistrement**

### 6.5.1 **Motivation**

La structure de tableau permet de traiter des données composées de plusieurs éléments tous de même type. Or d'autres données sont formées d'éléments de types différents comme par exemple:

- les dates chronologiques (année, mois, jour)
- les fiches bibliographiques (titre du livre, auteur, date de parution, ISBN...)
- les fiches personnelles (nom, prénom, âge, sexe, taille ...)
- ...

La nature différente de ces éléments conduit le programmeur à utiliser une structure permettant la définition explicite de chacun de ces éléments: les enregistrements.

[ Retour à la table des matières ]

### 6.5.2 **Généralités**

Un enregistrement (**record**) est formé d'un ou de plusieurs **éléments** (ou **champs**) pouvant être de **types différents.**

**Exemples:**

```
- record      (* notes d'une branche, cf. 6.3.2 *)
     notes : t_notes;
     moyenne : real;
     suffisant : boolean;
  end

- record     (* nombre complexe *)
```

```
        partie_reelle : real;
        partie_imaginaire : real;
    end
```

Les types enregistrement se déclarent ainsi:

```
    type t_branche = record (* notes d'une branche *)
                       notes : t_notes;
                       moyenne : real;
                       suffisant : boolean;
                     end;

        t_nombre_complexe = record (* nombre complexe *)
                              partie_reelle : real;
                              partie_imaginaire : real;
                            end;
```

Des variables de ces types se déclarent comme toujours:

```
   var maths : t_branche;

       nombre_1,
       nombre_2 : t_nombre_complexe;
```

Les types enregistrement font partie des types structurés (cf. annexe F). Concernant les champs il faut savoir que ceux-ci peuvent être de **n'importe quel type** (sauf un type fichier, cf. 10.3).

Mais:

- Il n'existe pas de **constante** d'un type enregistrement.
- Il n'existe pas d'**opération** (autre que l'affectation et le passage comme paramètre) sur les enregistrements.
- Les seules **expressions** d'un type enregistrement sont les variables de ce type.
- Il n'y a pas de **fonction** (même prédéfinie) à résultat d'un type enregistrement.
- Il n'y a pas d'**entrées-sorties** sur les enregistrements.

Ces limitations rendent les enregistrements peu utilisables tels quels. Dans la majorité des cas leur utilisation réside dans l'utilisation de leurs champs.

[ Retour à la table des matières ]

### 6.5.3 **Accès aux champs d'un enregistrement**

Comment accéder aux champs d'un enregistrement? Un tel champ est défini par le nom de l'enregistrement ainsi que par son nom propre.

**Exemples:**

- `maths.moyenne` représente le champ *moyenne* de l'enregistrement *maths*
- `maths.notes` représente le champ *notes* de l'enregistrement *maths*

```
                              à savoir un tableau!

   - maths.notes [ 1 ]        représente le premier élément du champ notes de
                              l'enregistrement maths

   - nombre_1.partie_reelle   représente le champ partie_reelle de nombre_1
```

L'accès à un champ se fait donc en nommant l'enregistrement puis le champ et en séparant les deux noms par un point "." . Le résultat de cette construction est un champ du type spécifié à la déclaration du champ, utilisable de manière **absolument identique à une variable de ce type.**

Donc partout où une variable d'un type (simple) donné peut être utilisée, un champ de ce type peut l'être également.

[ Retour à la table des matières ]

### 6.5.4 **Affectation**

L'affectation se fait de manière habituelle entre **enregistrements de même type:**

```
nombre_1 := nombre_2;      (* cf. 6.5.2 *)
```

[ Retour à la table des matières ]

## 6.6 **L'instruction with**

Considérons le morceau de programme suivant:

> **var** nombre : t_nombre_complexe; (* pour le type *t_nombre_complexe* cf. 6.5.2 *)
> ...
> nombre.partie_reelle := 4.0;
> nombre.partie_imaginaire := - 2.0;

L'utilisation des champs d'un même enregistrement nécessite la répétition parfois fastidieuse de son nom. Ceci peut être évité grâce à l'instruction **with**:

```
with nombre do
begin
   partie_reelle := 4.0;
   partie_imaginaire := - 2.0;
end; (* with *)
```

Cette écriture allège le programme et peut améliorer sa lisibilité et l'efficacité du compilateur. Cependant il existe des cas où cette lisibilité est diminuée, principalement dans le cas d'instructions **with** imbriquées.

La forme générale est:

> ***with variable_enregistrement do instruction;***

[ Retour à la table des matières ]

# Index

[ Retour à la table des matières ]

# BIBLIOGRAPHIE


| | |
|---|---|
| Gottfried B. S. | "Programmation Pascal"<br>McGraw-Hill, série Schaum, 1986 |
| Guillemot J.-C. | "Initiation Pascal"<br>Ed. Radio Paris, 1983 |
| Keller A. M. | "Un premier cours de programmation en Pascal"<br>McGraw-Hill, 1985 |
| Lecarme O. & Nebut J.-L. | "Pascal pour programmeurs"<br>McGraw-Hill, 1985 |


[ Retour à la table des matières ]

**ANNEXE H**

# QUELQUES PROCEDURES PASCAL MACINTOSH UTILES

---

Les quelques procédures ci-dessous peuvent être utiles pour des programmes Pascal Macintosh:

- showtext; (* affiche la fenêtre de texte *)
- showdrawing; (* affiche la fenêtre de dessin *)
- hideall; (* ferme toutes les fenêtres *)
- move ( dx, dy ); (* comme *moveto* mais relatif à la position courante *)
- lineto ( x, y ); (* comme *line* mais le déplacement est absolu *)
- drawchar ( ch ); (* dessine le caractère contenu dans *ch* dans la fenêtre de *)
  (* dessin *)
- note ( frquence, amplitude, duree );

(* crée un son.
(* La fréquence est un entier dans l'intervalle 12.783360 [Hz] *)
(* L'amplitude est un entier dans l'intervalle 0..255 *)
(* La durée est un entier dans l'intervalle $0..255\ [1 \approx 0.02s]$ *)

[ Retour à la table des matières ]

**ANNEXE TP**

# TURBO PASCAL VERSION 6

---

Le but de cette annexe est de donner les principales différences entre Turbo Pascal et les notions du cours "Introduction à l'informatique" de l'EINEV. Chaque différence sera localisée grâce au numéro du paragraphe. Un texte explicatif décrira la situation de Turbo Pascal.

**Remarque générale :**

Toutes routines graphiques de Pascal Macintosh sont bien entendu inutilisables telles quelles en Turbo Pascal.

**Cas de Turbo Pascal**

| | |
|---|---|
| 2.5.3 | 63 caractères significatifs pour un identificateur |
| 3.5.2 | *maxint* vaut 32767 |
| 3.6.5 | par défaut une valeur entière est écrite sur le nombre minimal de positions, une valeur réelle sur 17 positions. |
| 4.1.2 | la variable de boucle **for** doit être locale au bloc englobant la boucle |
| 4.3.4 | pas de lecture d'une valeur booléenne |
| 7.1.5 | pas d'entrées-sorties sur des valeurs d'un type énuméré. |
| 7.2 | remarque 2 : si l'expression a une valeur ne correspondant à aucune des constantes mentionnées, l'instruction **case** est sautée. |
| 7.2 | remarque 3 : le mot réservé **else** est utilisé à la place de **otherwise** |
| 7.2 | il est possible d'utiliser aussi un intervalle borné par des constantes pour délimiter une branche d'une instruction **case** |
| 8.1.2 | un ensemble est limité à un maximum de 256 valeurs dont la représentation machine doit être comprise entre 0 et 255. |
| 8.2.2, 8.2.3, 8.2.4, 8.2.5, 8.2.6, 8.2.7 | Les différences sont assez nombreuses. Pour simplifier la lecture, ces paragraphes sont reproduits ci-après dans leur totalité après modification. |

[ Cas Turbo Pascal ( suite ) | Retour à la table des matières ]

En Pascal le traitement des chaînes de caractères est un point où les versions du langage sont toutes différentes. Cependant, plusieurs d'entre-elles offrent des opérations de manipulation similaires, agréables à utiliser. Parmi ces versions mentionnons Pascal Macintosh, Pascal VAX, Turbo Pascal...

Nous allons exposé ici la situation de Turbo Pascal.

8.2.2 **Généralités (Turbo Pascal)**

Une chaîne de caractères est une suite de caractères définie par

- une **taille** maximale fixe
- une **longueur courante** variable jamais supérieur à la taille
- les caractères présents dans la chaîne

Une chaîne de caractères se déclare au moyen du mot réservé **string**.

**Exemples :**

- **string** [ 10 ] (* chaîne de taille égale à 10 *)
- **string** [ 1 ] (* la plus petite chaîne possible *)
- **string** [ 255 ] (* la plus grande chaîne possible *)

Les types chaîne se déclare ainsi :

```
const max_car_nom = 30;       (* il est rare d'avoir un nom de 30 lettres. *)
      max_car_prenom = 15;    (* pour le prénom principal seulement. *)
      max_ligne = 80;         (* longueur maximale d'une ligne de texte. *)

type t_nom = string [ max_car_nom ];   (* max_car_nom est la taille *)
     t_prenom = string [ max_car_prenom ];
     t_ligne =  string [ max_ligne ];
     t_longue_ligne =  string;         (* équivalent ici à string [ 255 ] *)
```

Pour les variables de ces types :

```
var nom_de_famille : t_nom;
    prenom_principal : t_prenom;
    ligne_courante : t_ligne;
```

Les **constantes** d'un type chaîne sont écrites entre apostrophes :

```
'Abcd'   de taille et de longueur courante 4
''       chaîne vide (taille = longueur courante = 0)
'x'      de taille et de longueur courante égales à 1.
         C'est aussi une constante caractère!
```

Les **opérations** possibles (en plus de l'affection et du passage en paramètre) sur les chaînes de caractères sont :

\+ concaténation de deux chaînes de n'importe quel type

= < <= <> > >= entre chaîne de n'împorte quel type

Les **expressions** sont réduites aux constantes, variables, concaténations de chaînes et fonctions prédéfinies à résultat d'un type de chaîne.

**Remarques :**

Les types chaîne de caractères font partie des types structurés ( cf. annexe F ).

[ Suivant | Précédent | Retour à la table des matières ]

### 8.2.3 Accès aux éléments d'une chaîne de caractères

Il est possible d'accéder aux caractères composant une chaîne par la même notation que celle utilisée pour les éléments d'un tableau. L'accès aux caractères non définis est permis, sans effet notable, mais déconseillé. L'accès à l'élément d'indice 0 permet de consulter la longueur courante comme la fonction *lengh* qui doit toujouts être utilisée!

**Exemple :**

```
var nom_de_famille : t_nom;          (* cf. 8.2.2 *)
...
    nom_de_famille := 'Dupont';      (* cf. 8.2.4 *)
```

alors

```
nom_de_famille [ 1 ] vaud 'D'
nom_de_famille [ 2 ] vaud 'u'
...
nom_de_famille [ 6 ] vaud 't'

nom_de_famille [ 7 ] est non initialisé.

nom_de_famille [ 0 ] vaut chr ( 6 ) .
```

[ Suivant | Précédent | Retour à la table des matières ]

### 8.2.4 Affection

L'affection est possible entre chaînes de **n'importe quels types** (!) et de longueur quelconque. En cas de dépassement de taille la chaîne est tronquée à droite.

**Exemples :**

```
var nom_de_famille : t_nom;        (* cf. 8.2.2 *)
...
    nom_de_famille := 'Dupont';    (* après affection, nom_de_famille est de *)
                                   (* longueur 6 *)
    nom_de_famille := '';          (* après affection, nom_de_famille est de *)
                                   (* longueur 0 *)
```

Si l'on écrit nom_de_famille := '01234567890123456789012345678 9abcde'; les caractères *abcde* sont tronqués.

[ Suivant | Précédent | Retour à la table des matières ]

### 8.2.5 Procédures et fonctions prédéfinies

Jusqu'à présent le maniement des chaînes est rudimentaire. Il va être grandement amélioré par les procédures et fonctions de manipulation, qui sont prédéfinies.

#### 8.2.5.1 Fonctions prédéfinies

Soient *s, s1, s2, ..., sn, sub* des chaînes de caractères et *indice* , *nb* des nombres entiers. On a

| | |
|---|---|
| length ( s ) | donne la longueur courante de *s* |
| concat ( s1, s2, ... sn ) | donne la concaténation des chaînes *si* . Il est possible de concaténer également des caractères |
| pos ( sub, s ) | donne la position du premier caractère de *sub* dans *s* si *sub* est une sous-chaîne de *s* , 0 sinon |
| copy ( s, indice, nb ) | donne une copie de la sous-chaîne de longueur *nb* commençant au caractère *s [ indice ]* |

**Exemples :**

```
const     max = 10;

type      t_chaine = string [ max ];

var       s,
           sub : t_chaine;

begin
  s := 'ABCD'
```

```
sub := '123';
```

alors

| | |
|---|---|
| length ( s ) | donne 4 |
| length ( sub ) | donne 3 |
| concat ( s, sub ) | donne la chaîne ABCD123 |
| concat ( s, '', sub ) | donne la chaîne ABCD123 |
| pos ( 'BC', s ) | donne 2 |
| pos ( sub, s ) | donne 0 |
| pos ( 'BC', 'ABCDABC' ) | donne 2 |
| copy ( s, 3, 2 ) | donne la chaîne CD |
| copy ( s, 2, 3 ) | donne la chaîne BCD |
| copy ( s, *indice* , 0 ) | correct pour tout valeur *indice* entière! Donne la chaîne vide '' |
| copy ( s, -1, 4 ) | correct! Donne la chaîne ABCD |
| copy ( s, 3, 4 ) | correct! Donne la chaîne CD |

**Remarques :**

1. Des caractères peuvent être utilisés dans ces fonctions
2. Une chaîne est tronquée si elle dépasse 255 caractères.

[ Suivant | Précédent | Retour à la table des matières ]

8.2.5.2 **Procédures prédéfinies**

Soient *s, sub* des chaînes de caractères et *indice, nb* des nombres entiers. On a

| | |
|---|---|
| delete ( s, indice, nb ) | enlève *nb* caractères dans *s* à partir du caractère *s [ indice ]* |
| insert ( sub, s, indice ) | insère la chaîne *sub* dans *s* à partir du caractère *s [ indice ]* |

**Exemples** ( cf. 8.2.5.1 pour les déclarations et initialisations ) :

| | |
|---|---|
| delete ( s, 1, 2 ); | *s* vaut alors la chaîne CD |
| delete ( s, 2, 1 ); | *s* vaut alors la chaîne ACD |
| delete ( s, -1, 4 ); | correct! *s* vaut alors la chaîne CD |
| delete ( s, 3, 4 ); | correct! *s* vaut alors la chaîne AB |
| insert ( sub, s, 3 ); | *s* vaut alors la chaîne AB123CD |
| insert ( sub, s, -1 ); | correct! *s* vaut la chaîne 123ABCD |
| insert ( sub, s, 5 ); | correct! *s* vaut la chaîne ABCD123 |

**Remarques :**

1. Des caractères peuvent être utilisés dans ces procédures !
2. La chaîne est tronquée à droite en cas de dépassement de taille.

[ Suivant | Précédent | Retour à la table des matières ]

### 8.2.6 **Entrées-sorties**

Turbo Pascal permet la lecture et l'écriture de chaînes de caractères au moyen des procédures prédéfinies *read* et *write* . Voici leur sématique :

read ( s ) ; lit les caractères de la ligne depuis la position courante

- jusqu'au caractère de fin de ligne **non** compris, puis affecte cette chaîne à la chaîne *s* (attention à la règle des longueurs, cf. 8.2.4), si le nombre de caractères à lire est inférieurs à *length ( s );*
- jusqu'à concurrence de *length ( s )* caractères, puis affecte cette chaîne à la châine *s,* si le nombre de caractères à lire est supérieur ou égal à *length ( s ).*

On peut donc lire une ligne complètement (y compris la marque de fin de ligne) par *readln ( s );* .

| | |
|---|---|
| write ( s ); | écrit la chaîne *s* sur *length ( s )* positions. |
| write ( s : nb ); | écrit la chaîne *s* sur *nb* positions. Mais<br>- si *nb* > *length ( s )*, la chaîne *s* est cadrée à droite<br>- si *nb* < *length ( s )*, seuls les *nb* premiers caractères sont écrits |

**Remarque :**

Les procédures *readln* et *writeln* sont naturellement utilisables avec les chaînes de caractères.

[ Suivant | Précédent | Retour à la table des matières ]

### 8.2.7 **Passage en paramètre et résultat de fonction**

Le passage en paramètre de chaînes de caractères peut s'effectuer comme habituellement. Il existe cependant une autre possibilité, celle qui est exposée maintenant.

Il est possible d'écrire

```
procedure exemple (     s1 :  string;
                    var s2 :  string );
...
```

Alors

- le paramètre *s1* est toujours de taille maximale 255 et héritera la longueur du paramètre effectif.
- le paramètre *s2* est toujours de taille maximale 255, héritera la longueur du paramètre effectif et exige un paramètre effectif de type **string**!

**Exemple :**

```
var   s : string
...
   s := '1234'
   exemple ( 'bac', s );
```

Le paramètre *s1* est alors de longueur 3 et vaut *' bac '* à l'appel de *exemple* .
Le paramètre *s2* est alors de longueur 4 et vaut *' 1234 '* à l'appel de *exemple.*

Le type du résultat d'une fonction peut également être mentionné comme **string**.

**Exemple :**

```
function exemple : string;
begin (* exemple *)
  exemple := 'abc'
end; (* exemple *)
...
s := exemple;                (* s  vaut alors ' abc' , est donc la longueur 3 *)
```

[ Suivant | Précédent | Retour à la table des matières ]

---

**Cas de Turbo Pascal ( suite )**

9.3.3 la fonction *ord* ne peut pas s'utiliser avec une variable d'un type pointeur

10.2 Turbo Pascal permet l'accès direct

10.3.2 Turbo Pascal exige la fermeture explicite des fichiers par la procédure *close* :

***close ( nom_variable_fichier );***

10.3.3 il n'est pas possible d'accéder à la fenêtre par la notation avec ^, *get* et *put* n'existent pas. Il faut utiliser les deux procédures *read* et *write* ( ! ). Le deuxième exemple du § 10.3.3 devient :

```
read ( bibliotheque, livre );
...
write ( bibliotheque, livre );
```

10.3.4 *put* et *get* n'existent pas. On a par contre

read ( exemple, donnee ); lit dans *donnee* l'élément suivant du fichier *exemple*

write ( exemple, donnee ); écrit la valeur de *donnee* à la fin du fichier *exemple.*

10.3.5 il n'y a pas de fichiers internes en Turbo Pascal. L'association entre une variable fichier et un fichier externe doit se faire **avant** l'utilisation de *reset* ou *rewrite* par la procédure *assign :*

***assign ( nom_variable_fichier, chaine_de_caractères );***

où *nom_variable_fichier* est le nom d'une variable d'un type de fichier
*chaine_de_caracteres* représente le nom externe du fichier avec indication du périphérique et du chemin d'accès si nécessaire.

10.3.6 L'exemple complet devient le suivant :

[ Cas Turbo Pascal ( suite ) | Cas Trubo Pascal ( précédent ) | Retour à la table des matières ]

Voici un programme permettant de copier un fichier dans un autre.

```
program copie_de_fichiers ( input, output, original, copie );
(* Auteur : ... *)
    const  long_max_nom = 30;  (*nombre maximal de caractères du nom d'un auteur *)

           fichier_a_copier = 'exemple.dat';  (* nom du fichier d'origine *)

           fichier_copie = 'resultat.dat';    (* nom du fichier qui contiendra la copie
    *)
                                               (* effectuée par le programme *)

    type   t_auteur_livre =  string [ long_max_nom ];  (* syntaxe Turbo Pascal! *)

           t_livre_cote =  record        (* représente une fiche bilbiographique *)
                             nom_auteur : t_auteur_livre   (* nom de l'auteur *)
                             cote : integer;               (* cote en bibliothèque *)
                             annee : integer;              (* année de parution *)
                           end;

           t_livre_biblio =  file of t_livre_cote;

    var    original : t_fichier_biblio;      (* fichier à copier *)
           copie : t_fichier_bilbio;          (* copie à créer *)
           element : t_livre_cote;            (* élément lu et copié *)
begin   (* copie_de_fichiers *)
    assign ( original, fichier_a_copier );    (* associer les fichiers externes *)
    assign ( copie, fichier_copie );

    reset ( original );    (* ouvrir le fichier à copier *)
    rewrite ( copie );     (* initialiser la copie *)

    while note eof ( original ) do  (* parcourir les éléments du fichier à copier *)
    begin
        read ( original, element );   (* lire l'élément courant *)
        write ( copie, element );     (* écrire l'élément courant *)
    end;

    close ( original );   (* NE PAS OUBLIER *)
    close ( copie );      (* NE PAS OUBLIER *)
end.
```

[ Suivant | Précédent | Retour à la table des matières ]

---

**Cas de Turbo Pascal ( suite )**

| | |
|---|---|
| 10.4.1 | la remarque 3 (*real* et *write* sont des abréviations pour ...) ne s'applique pas |
| 10.4.2 | la marque de fin de fichier peut être tapée au moyen de <ctrl> z |

| | |
|---|---|
| 10.4.2 | l'instruction *read ( exemple, caractere );* a l'effet suivant si le caractère à lire est la marque de fin de ligne :<br>caractere := chr ( 13 ); (* caractère *cr* *)<br>readln ( exemple ); |
| 12.1.2 | l'accès à une composante non existante d'un enregistrement à variante n'est pas contrôlé, donc possible, mais à éviter ! |
| 12.1.3 | la remarque 5 (formes de *new* et *dispose* sur des variables dynamiques représentant des enregistrements à variante) ne s'applique pas. |
| 12.3 , 12.3.1, 12.3.2, 12.3.3 | le passage de procédures et fonctions en paramètre est possible, mais avec un autre mécanisme non décrit ici. |

[ Cas Trubo Pascal ( précédent ) | Retour à la table des matières ]

Annexe C : Turbo Pascal possède en plus les mots réservés suivants (!!) :

**absolute, boolean, byte, char, comp, double, extended, implementation, inline, integer, interface, interrupt, longint, real, shl, shortint, shr, single, string, subrange, unit, uses, word, xor**

Annexe E : Les routines suivantes n'existent pas :

get, pack, put, unpack, new (partie variante), dispose (partie variante)

**ANNEXE EX P**

# EXEMPLES DE PROGRAMMES PASCAL

---

Cette annexe contient plusieurs exemples de programmes écrits en Pascal et fonctionnant sur Macintosh. Ces programmes sont conçus de telle manière que :

- chacun illustre spécialement l'utilisation d'un (ou de plusieurs) point du langage Pascal
- leur style soit un très bon exemple de ce qu'il faut faire pour obtenir des programmes clairs et lisibles.

Dans chaque exemple une note mentionne les notions de Pascal nécessaires pour comprendre le programme. Ces notes désignent les paragraphes ou chapitres du cours "Introduction à l'informatique" utilisés; elles se présentent sous la forme du commentaire

(* Pascal utilisé : jusqu'à la fin du ..... *)

**Sommaire**



[ Retour à la table des matières ]

---

**EXEMPLE 1**

```
(*************************************************************************)
(*                                                                       *)
(* Nom du fichier          : GEN_ALEA.PAS                                *)
(* Auteur(s)               : J. Skrivervik     EI-6                      *)
(* Date                    : le 21.5.1987                                *)
(*                                                                       *)
(* But                     : Créer un générateur de nombre aléatoires.   *)
(*                           Ce générateur nous affichera un nombre      *)
(*                           compris entre 0 et 1.                       *)
(*                                                                       *)
(*                                                                       *)
(* Date de modification    : 24.7.1987 PBT                               *)
(* Raison de la modif.     : soigner le style                            *)
(* Module(s) appelé(s)     :                                             *)
(* Matériel particulier    :                                             *)
(* Modalités d'exécution   :                                             *)
(*                                                                       *)
```

```
(*********************************************************************)

program gen_alea ( input,output );
(* Programme pouvant être aisément transformé en une fonction. *)
(* Pascal utilisé : jusqu'à la fin du paragraphe 4.1 *)
    const  pi = 3.14159;          (* valeur de pi ( 6 chiffres significatifs) *)
           max = 10;               (* quantité de nombres aléatoires affichés *)

    var    aleatoir : real;        (*nombre aléatoire (résultat)              *)
           tempo : real;           (* variable de travail temporaire           *)
           compteur : integer;     (* variable de boucle                       *)
begin  (* gen_alea *)

     showtext;
     (* présentation du programme et lecture des données *)
     writeln ('Bonjour. Je calcule', max : 4,' nombres aléatoires');

     write ('Entrez un nombre entre 0 et 10 : ' : 40 );
     readln (aleatoire);            (* valeur à partir de laquelle sera calculé le
*)
     writeln;                       (* premier nombre aléatoire *)

     for compteur := 1 to max  do   (* on affiche  max  nombres *)
     begin
         tempo := aleatoire + pi     (* début du calcul d'un nombre aléatoire *)
         tempo := exp ( 5.0 * ln ( tempo ) );
         aleatoire := tempo - trunc ( tempo );
         writeln (aleatoir : 40 : 4 );
         writeln;
     end;

end.
```

---

[ Suivant | Précédent | Sommaire | Retour à la table des matières ]

---

**EXEMPLE 2**

```
(*********************************************************************)
(*                                                                   *)
(* Nom du fichier          : TABLE_PYTHA.PAS                         *)
(* Auteur(s)               : J. Skrivervik     EI-6                  *)
(* Date                    : le 21.5.1987                            *)
(*                                                                   *)
(* But                     : Créer une table de Pythagore, c'est-à-dire *)
(*                           une table ou l'intersection entre ligne et *)
(*                           colonne représente la multiplication des   *)
(*                           deux                                    *)
(*                                                                   *)
(* Date de modification    : 24.7.1987 PBT                           *)
(* Raison de la modif.     : soigner le style                        *)
(* Module(s) appelé(s)     :                                         *)
(* Matériel particulier    :                                         *)
(* Modalités d'exécution   :                                         *)
(*                                                                   *)
```

```
(*********************************************************************)

program table_pytha ( input, output );
(* Pascal utilisé : jusqu'à la fin du paragraphe 4.1 *)
    var    nombre_ligne_col : integer;    (* nombre de lignes ( ou de colonnes ) que *)
                                          (* contiendra la table *)
           i, j : integer;                (* variables de boucle *)
begin      (* table_pytha *)

     showtext;
     (* présentation du programme et lecture des données *)
     writeln ('Bonjour. Je calcule une table de Pythagore. Cette table est carrée.');
     write ('Quel nombre de lignes ( donc de colonnes ) désirez-vous ? :       ');
     readln ( nombre_ligne_col );

     writeln;
     writeln ('Table de Pythagore' : 50 );  (* titre de la table produite comme
résultat *)
     writeln ('------------------' : 50 );
     writeln;

     for i := 1 to nombre_ligne_col do       (* boucle traitant les lignes *)
     begin
         for  j:= 1 to  nombre_ligne_col do  (* boucle traitant les colonnes *)
                 write ( i * j : 6 );          (* calcul d'un élément de la table *)

         writeln;                       (* écriture d'une ligne de la table *)
         writeln;                       (* on laisse une ligne vide entre chaque
ligne *)
     end;

end
```

---

[ Suivant | Précédent | Sommaire | Retour à la table des matières ]

---

**EXEMPLE 3**

```
(*********************************************************************)
(*                                                                   *)
(* Nom du fichier          : PGCD.PAS                                *)
(* Auteur(s)               : J. Skrivervik     EI-6                  *)
(* Date                    : le 2.6.1987                             *)
(*                                                                   *)
(* But                     : Programme calculant le plus grand commun*)
(*                           diviseur de 2 nombres entiers           *)
(*                                                                   *)
(*                                                                   *)
(*                                                                   *)
(* Date de modification    : 24.7.1987 PBT                           *)
(* Raison de la modif.     : soigner le style                        *)
(* Module(s) appelé(s)     :                                         *)
(* Matériel particulier    :                                         *)
(* Modalités d'exécution   :                                         *)
```

```
(*                                                                    *)
(**********************************************************************)

program pgcd ( input, output );
(* Pascal utilisé : jusqu'à la fin du paragraphe 4.4 *)
    var  nombre_1 : integer;        (* variable qui va contenir le premier nombre *)
         nombre_2 : integer;        (* varibale qui va contenir le deuxième nombre *)
begin    (* pgcd *)

     showtext;
     writeln;                        (* présentation du programme *)
     writeln ('Programme calculant le PGCD de deux nombres entiers' : 65 );
     writeln;

     writeln;                        (* lecture des données *)
     write ('Valeur du premier nombre :   ' : 40 );
     readln ( nombre_1 );
     writeln;
     write ('Valeur du deuxième nombre :  ' : 40 );
     readln ( nombre_2 );
     writeln;

     (* calcul du PGCD *)
     while nombre_1 <> nombre_2 do  (* PGCD trouvé lorsqu'ils sont égaux *)
        if nombre_1 > nombre_2 then (* soustraire le plus petit du plus grand *)
              nombre_1 := nombre_1 - nombre_2
        else nombre_2 := nombre2_ - nombre_1;

     writeln;
     writeln ('Le PGCD des deux nombres est : ' : 60, nombre_1 : 5 );

end.
```

---

[ Suivant | Précédent | Sommaire | Retour à la table des matières ]

---

**EXEMPLE 4**

```
(**********************************************************************)
(*                                                                    *)
(* Nom du fichier          : CONVERSIONS.PAS                          *)
(* Auteur(s)               : J. Skrivervik     EI-6                   *)
(* Date                    : le 20.5.1987                             *)
(*                                                                    *)
(* But                     : Donner soit le code ASCII d'un caractère,  *)
(*                           soit le caractère à partir de son code   *)
(*                           ASCII                                    *)
(*                                                                    *)
(*                                                                    *)
(* Date de modification    : 24.7.1987 PBT                            *)
(* Raison de la modif.     : soigner le style                         *)
(* Module(s) appelé(s)     :                                          *)
(* Matériel particulier    :                                          *)
(* Modalités d'exécution   :                                          *)
```

```
(*                                                                    *)
(**********************************************************************)

program conversions ( input, output );
(* Pascal utilisé : jusqu'à la fin du chapitre 4 *)
    const   plus_grand_code = 127;    (* valeur limite du code ASCII standard *)
            premier_car_imp = '';     (* l'espace est le premier caractère imprimable *)
            dernier_car_imp = '~';    (* le tilde est le dernier caractère imprimable *)
            en_ascii = 1;             (* converstion en ASCII *)
            en_caractere = 2;         (* converstion en caractère *)

    var  caractere : char;           (* caractère lu *)
         nombre : integer;           (* nombre lu *)
         choix : integer;            (* choix du type de conversion *)
         fin : boolean;              (* pour l'arrêt du programme *)
begin   (* conversions *)

     showtext;
     writeln;              (* présentation du programme *)
     writeln,
     writeln ('Ce programme donne le code ASCII d'' un caractere, ou le caractere');
     writeln ('a partir d''un code ASCII.');
     writeln;
     fin := false;
     while not fin  do    (* tant que l'utilisateur veut continuer *)
     begin
          writeln;
          writeln;

          (* on répète jusqu'au bon choix de l'utilisateur *)
          choix := 0;
          while ( choix <> en_ascii ) and ( choix <> en_caractere ) do
          begin
              write ('Conversion en ASCII < 1 >, conversion en caractere < 2 > : ' :
65 );
              readln ( choix );
          end;

          if choix = en_ascii then
          begin                     (* conversion d'un caractère en code ASCII *)
              write ('Introduisez un caractere : ' : 65 );
              read ( caractere );
              writeln;
              writeln ('Le code ASCII de ', caractere, 'est : ', ord ( caractere ) :
4 );
          end
          else        (* conversion d'un code ASCII en caractere *)
              begin
                  write ('Introduisez un code ASCII : ' : 65 );
                  readln ( nombre );
                  writeln;

                  (* le nombre peut-il être converti ? *)
                  if ( nombre >= 0 ) and ( nombre <= plus_grand_code ) then

                       (* le nombre peut-il être affiché ? *)
                       if ( nombre >= ord ( premier_car_imp ) ) and
```

```
                        ( nombre <= ord ( dernier_car_imp ) ) then
                    begin
                    write ('Le caractere ayant pour code ASCII', nombre : 4 );
                      writeln ('est :', chr ( nombre ) );
                    end
                    else  begin
                                write ('Le caractere correspondant est un caractere
de controle');
                                writeln ('non imprimable');
                             end
              else
                     writeln ('Nombre négatif ou trop grand pour être converti
!!!!!!!');
            end;

         writeln;    (* l'utilisateur veut-il continuer ou terminer ? *)
         write ('Voulez-vous continuer ( oui = O, non = N ) ? : ' : 65 );
         read ( caractere );

         fin := ( caractere = 'n' ) or ( caractere = 'N' );  (* condition de fin de
programme *)

    end;

end.
```

---

[ Suivant | Précédent | Sommaire | Retour à la table des matières ]

---

**EXEMPLE 5**

```
(*************************************************************************)
(*                                                                       *)
(* Nom du fichier          : ARCSIN.PAS                                  *)
(* Auteur(s)               : J. Skrivervik     EI-6                      *)
(* Date                    : le 11.6.1987                                *)
(*                                                                       *)
(* But                     : Construire une fonction acrsinus  donnant   *)
(*                           l'angle en radians à partir de son sinus    *)
(*                                                                       *)
(*                                                                       *)
(*                                                                       *)
(* Date de modification    : 24.7.1987 PBT                               *)
(* Raison de la modif.     : soigner le style                            *)
(* Module(s) appelé(s)     :                                             *)
(* Matériel particulier    :                                             *)
(* Modalités d'exécution   :                                             *)
(*                                                                       *)
(*************************************************************************)

program arcsin ( input, output );
(* ce programme contient et utilise une fonction : la fonction arcsinus *)
(* Pascal utilisé : jusqu'à la fin du chapitre 5 *)
    const  pi = 3.14159;

    var  nombre : real;   (* nombre dont on veut calculer l'arcsinus *)
```

```
    function arcsinus ( sinus : real ) : real;
    (* cette fonction nous donne les angles en radians *)
    (* Paramètre : sinus  valeur d'un sinus à partir de laquelle on cherche l'angle *)

         const   zero = 0.0    (* trois sinus particuliers *)
                 moins_un = -1.0;
                 plus_un = 1.0;

    begin  (* arcsinus *)
         (* il existe 3 cas particuliers et un cas général *)

         if sinus = zero then                  (* 1er cas particulier *)
            arcsinus := 0.0

         else if sinus = plus_un then          (* 2e cas particulier *)
                 arcsinus := pi / 2.0

               else if sinus = moins_un then   (* 3e cas particulier *)
                        arcsinus := - pi / 2.0

                    else                        (* cas général *)
                         arcsinus := arctan ( sinus / sqrt ( 1.0 - sqr ( sinus ) ) );
    end;   (* arcsinus *)
begin (* arcsin *)

     showtext;
     (* présentation du programme *)
     writeln;
     writeln;
     writeln ('Programme permettant de donner un angle en radians a partir de son
sinus');
     writeln;
     writeln;

     (* lecture des données *)
     write ('Donner la valeur de votre sinus ( entre -1.0 et 1.0 ) : ' : 40 );
     readln ( nombre );
     writeln;

     nombre := arcinus ( nombre );       (* calcul de l'angle à l'aide de la *)
                                         (* fonction arcsinus *)

     (* écriture des résulats *)
     writeln;
     writeln ('L''arcsinus de votre sinus est : ' : 40, nombre : 9 : 5 );

end.
```

---

[ Suivant | Précédent | Sommaire | Retour à la table des matières ]

---

**EXEMPLE 6**

```
(*****************************************************************)
(*                                                               *)
```

```
(* Nom du fichier          : RES_EQUATION_2.PAS                        *)
(* Auteur(s)               : J. Skrivervik     EI-6                    *)
(* Date                    : le 30.5.1987                              *)
(*                                                                     *)
(* But                     : Programme permettant de résoudre une équa- *)
(*                           tion du 2ème degré ( solutions réelles et  *)
(*                           complexes                                 *)
(*                                                                     *)
(*                                                                     *)
(* Date de modification    : 25.7.1987 PBT                             *)
(* Raison de la modif.     : soigner le style, correction d'erreurs    *)
(* Module(s) appelé(s)     :                                           *)
(* Matériel particulier    :                                           *)
(* Modalités d'exécution   :                                           *)
(*                                                                     *)
(***********************************************************************)

program res_equation ( input, output );
(* Pascal utilisé : jusqu'à la fin du paragraphe 7.2 *)
(* Nous avons ici 5 cas différents qui peuvent se présenter : l'équation du 2ème
degré *)
(* peut être soit dégénérée, de degré 1, avoir des solutions réelles ou complexes. *)
(* distinctes ou complexes *)
    const nb_positions = 15;     (* colonnes utilisées pour écrire les solutions *)

    type  t_cas_possible = ( degeneree, degre_1, relle, double, complexe );

    var   cas : t_cas_possible;     (* 5 cas possibles *)
          discriminant : real;       (* b2 - 4ac *)
          coef_degre_2 : real;       (* coefficient du terme du 2e degré *)
          coef_degre_1 : real;       (* coefficient du terme de degré 1 *)
          coef_degre_0 : real;       (* coefficient du terme de degré 0 *)
          x1, x2 : real;             (* 2 solutions *)
          reel, imaginaire : real;   (* en cas de solutions complexes *)
begin   (* res_equation *)

     showtext;
     writeln;                         (* présentation du programme *)
     writeln ('Programme de resolution d'equation du 2e degre');

     writeln;                         (* lecture des coefficients *)
     write ('Coefficient du terme de degre 2 : ' : 50 );
     readln ( coef_degre_2 );
     write ('Coefficient du terme de degre 1 : ' : 50 );
     readln ( coef_degre_1 );
     write ('Coefficient du terme de degre 0 : ' : 50 );
     readln ( coef_degre_0 );

     (* détermination des différents cas possible *)
     if coef_degre_2 = 0 then                        (* degré 1 ou dégénérée *)
         if coef_degre_1 = 0 then  cas := degeneree
         else cas := degre_1

     else                                            (* degré 2 *)
         (* calcul du discriminant *)
         discrimant := sqr ( coef_degre_1 ) - 4.0 * coef_degre_2 * coef_degre_0;
```

```
        if discriminant = 0 then cas := double        (* racines réelles confondues
*)
        else    if discriminant > 0 then cas := relle (* racines réelles distinctes
*)
             else cas := complexe;                     (* racines complexes *)

    (* calcul des racines dans chacun des cas *)
    case cas of

         degenere:           (* cas dégénéré *)
                   if coef_degre_0 <> 0 then  writeln ('IMPOSSIBLE' : 35 )
                   else  writeln ('INDETERMINE' : 35 );

          degre_1 : begin                                (* cas degré 1 *)
                        write ('solution : X = ');
                        writeln ( - coef_degre_0 / coef_degre_1 : nb_positions : 5
);
                    end;

          double :  begin      (* cas racines confondues *)
                        write ('solution double : X1 = X2 = ' : 35 );
                        writeln ( - coef_degre_1 / ( coef_degre_2 * 2.0 ) :
nb_positions : 5 );
                    end;

          reelle :  begin      (* cas racines distinctes *)
                        X1 := ( - coef_degre_1 + sqrt ( discriminant ) / (
coef_degre_2 * 2.0 );
                        X2 := ( - coef_degre_1 - sqrt ( discriminant ) / (
coef_degre_2 * 2.0 );
                        writeln ('solutions : X1 = ' : 35, X1 : nb_positions : 5 );
                        writeln ('X2 = ' : 35, X2 : nb_positions : 5 );
                    end;

          complexe : begin     (* cas racines complexes *)
                         reel := - coef_degre_1 / ( coef_degre_2 * 2.0 );
                         imaginaire := sqrt ( - discriminant ) / ( coef_degre_2 *
2.0 );
                         write ('solutions : X1 = ' : 35, reel : nb_positions : 5 );
                         writeln (' + i', imaginaire : nb_positions : 5 );
                         write ('X2 = ' : 35, reel : nb_positions : 5 );
                         writeln (' -i', imaginaire : nb_positions : 5 );
                     end;

      end;   (* case *)

end.
```

---

[ Suivant | Précédent | Sommaire | Retour à la table des matières ]

---

**EXEMPLE 7**

```
(*****************************************************************)
(*                                                               *)
(* Nom du fichier          : PUISSANCE_REC.PAS                   *)
```

```
(* Auteur(s)               : J. Skrivervik     EI-6                    *)
(* Date                    : le 4.6.1987                               *)
(*                                                                     *)
(* But                     : Programme permettant de calculer la puis-  *)
(*                           sance entière, positive ou négative d'un   *)
(*                           nombre réel.                               *)
(*                                                                     *)
(*                                                                     *)
(* Date de modification    : 25.7.1987 PBT                             *)
(* Raison de la modif.     : soigner le style                          *)
(* Module(s) appelé(s)     :                                           *)
(* Matériel particulier    :                                           *)
(* Modalités d'exécution   :                                           *)
(*                                                                     *)
(***********************************************************************)

program puissance_recursive ( input, output );
(* Pascal utilisé : jusqu'à la fin du chapitre 11 *)
    var nombre : real;     (* nombre qui va être élevé à la puissance exposant *)
         exposant : integer (* la puissance considérée *)

    function puissance ( nombre : real; exposant : integer ) : real;
    (* fonction permettant de calculer une puissance de manière récursive *)
    (* Paramètres : nombre    nombre ( <> 0 ) qui va être élevé à une certaine puissance
    *)
                    exposant  puissance considérée *)

    begin  (* puissance *)

         if exposant = 0 then           (* condition d'arrêt de l'appel récursif *)
              puissance := 1.0

         else if exposant > 0 then     (* exposants positifs *)
                    puissance := puissance ( nombre, exposant - 1 ) * nombre

               else                      (* exposants négatifs *)
                    puissance := puissance ( nombre, exposant + 1 ) / nombre;

    end;    (* puissance *)
begin   (* puissance_recursive *)

     showtext;
     writeln;              (* présentation du programme *)
     writeln;
     writeln ('Programme permettant d'elever un nombre a une puissance positive ou
negative' : 65 );
     writeln;

     writeln;              (* lecture des données *)
     write ('Nombre a elever a une certaine puissance : ' : 45 );
     readln ( nombre );
     writeln;

     if nombre <> 0.0 then (* on demande la puissance seulement s'il faut calculer!
*)
     begin
         write ('Puissance considérée : ' : 45 );
```

```
        readln ( exposant );
        writeln;
        writeln;

        (* fin du dialogue avec l'utilisateur. Ecriture des résultats *)
        write ('SOLUTION : ' : 45, puissance ( nombre, exposant ) : 15 : 5 );
    end
    else writeln ('Aucun interet d'elever 0 a une certaine puissance!');

end.
```

---

[ Précédent | Sommaire | Retour à la table des matières]