# Chapitre 2 : Programmer avec un langage

## 2.1.Les langages

- Historique des langages de programmation
- Langages procéduraux
- langages fonctionnels
- langages logiques
- langages objets
- langages de spécification
- langages hybrides

## 2.2.Relations binaires

- Rappel et conventions
- matrice d'une relation binaire
- fermeture transitive d'une relation binaire

### 2.3.Théorie des langages

- notations et définitions
- grammaire formelle
- classification de Chomsky
- applications - exemples

## 2.4.Les bases du langage Delphi-Pascal

- structure d'un programme
- les opérateurs
- déclarations des types
- instructions
- fonctions/procédures
- paramètres
- visibilité
- passage par adresse
- variables dynamiques
- récursivité

# 2.1 : Les langages

**Plan du chapitre:**

## 1.Historique des langages

# 1. Historique des langages de programmation

La communication entre l'homme et la machine s'effectue à l'aide de plusieurs moyens physiques externes. Les ordres que l'on donne à l'ordinateur pour agir sont fondés sur la notion d'instruction comme nous l'avons déjà vu. Ces instructions constituent un langage de programmation. Depuis leur création, les langages de programmation ont évolué et se sont diversifiés.

Schématiquement il est possible de les classer en cinq catégories :

**1° Les langages procéduraux ou impératifs.**
**2° Les langages fonctionnels.**
**3° Les langages logiques.**
**4° Les langages objets.**
**5° Les langages de spécification.**

L'un des principaux objectifs d'un langage de programmation est de permettre la construction de logiciels ayant un minimum de qualités comme la fiabilité, la convivialité, l'efficacité.
Il faut connaître l'histoire des langages et se rendre compte qu'à ce jour, malgré les nouveaux langages du marché et leur efficacité, c'est **Cobol** qui est le plus utilisé (numériquement 200 milliards de lignes Cobol seraient intégrées à des applications existantes [programmez, n°63 Avril 2004] dont 5 milliards de lignes nouvelles chaque année) dans le monde.

L'investissement intellectuel et matériel prédomine sur la nouveauté. Cette remarque est la clef de la compréhension de l'évolution actuelle et future des langages.

Les langages ont fait leurs premiers pas directement sur des instructions machines écrites en binaire, donc rudimentaires sur le plan sémantique. Les améliorations sur cette catégorie de langages se sont limitées à construire des langages symboliques (langage avec mnémonique) et des macro-assembleurs. J.Backus d'IBM avec son équipe a mis au point dès 1956-1958 le premier langage évolué de l'histoire, uniquement conçu pour le calcul scientifique (à l'époque l'ordinateur n'était qu'une calculatrice géante).

Les années 70 ont vu s'éloigner un rêve d'informaticien : parler et communiquer en langage naturel avec l'ordinateur.
Actuellement les langages évolués se diversifient et augmentent en qualité d'abstraction et de convivialité.

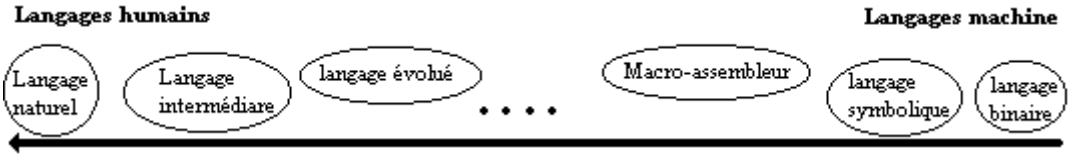



*fig : classification sur un axe d'abstraction : de la machine à l'homme*

Les langages majoritairement les plus utilisés actuellement sont ceux qui font partie de la catégorie des langages procéduraux ou Hybrides. Les ordinateurs étant des machines de Turing (améliorées par von Neumann), la notion de mémoire machine est représentée par la

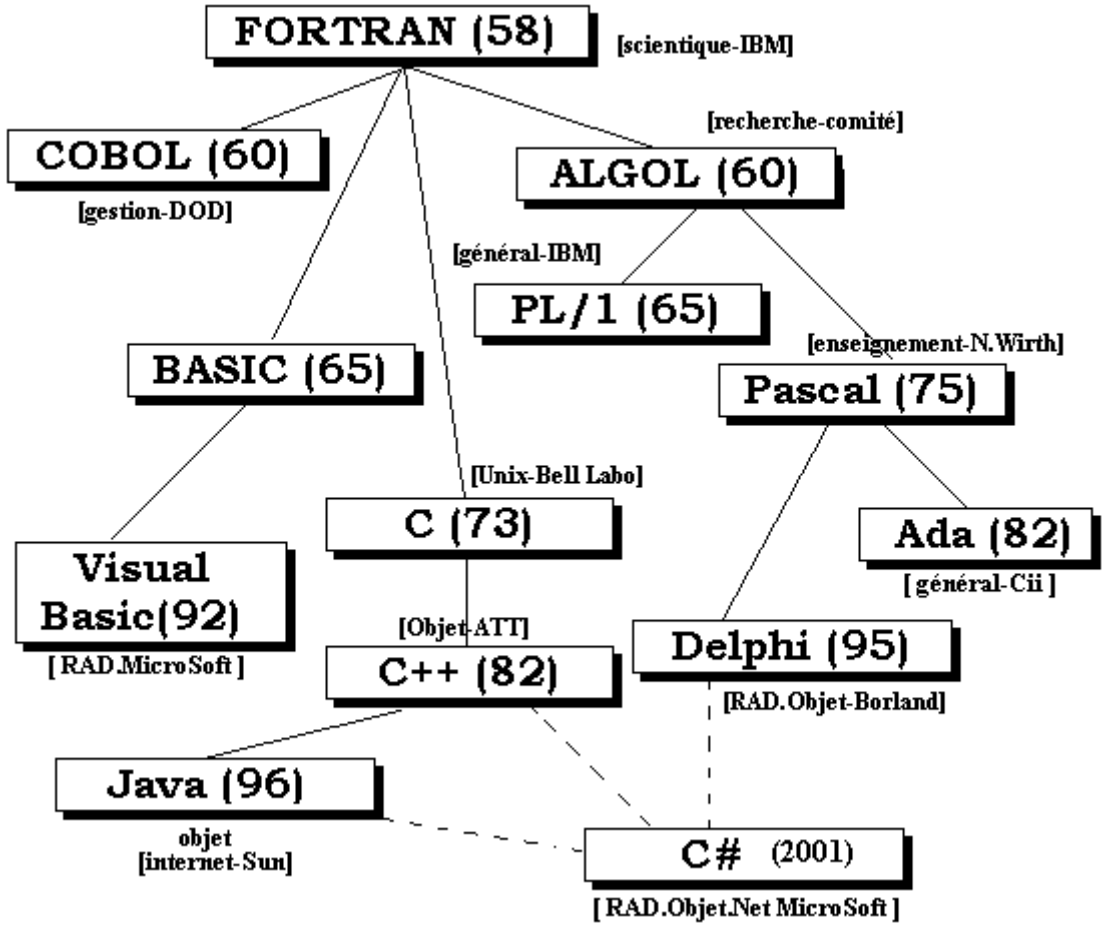
donnée abstraite qu'est une variable, dans un langage procédural. D'autre part, les machines de Türing sont séquentielles et les langages impératifs traitent les instructions séquentiellement. Ceci indique que les langages procéduraux sont parfaitement bien adaptés à l'architecture de l'ordinateur ; ils sont donc plus " facilement " adaptables à la machine.

### *1.1 Les langages procéduraux ou impératifs*

Tous les langages procéduraux ont un ancêtre commun : le langage **FORTRAN**.

Voici un arbre généalogique (non exhaustif) de certains langages connus. Pour chaque langage nous avons indiqué quelques éléments de référence.

Par exemple : **FORTRAN (58) [scientifique - IBM]** signifie que le premier compilateur commercial a été diffusé environ en 1958, que le domaine d'activité pour lequel le langage a été élaboré est le domaine du calcul scientifique, enfin qu'il s'agit d'un acte commercial puisque c'est la compagnie IBM qui l'a fait réaliser.



Dans cette courte liste, seuls **Algol**, **Basic** et **Pascal** sont des langages qui ont été conçus par des équipes dans des buts de recherche ou d'enseignement. Les autres langages sont élaborés par des firmes et des compagnies dans des buts de commercialisation, de rationalisation des coûts de gestion (DOD) etc... Les langages de programmation, comme le reste des outils de la science informatique, sont fortement soumis aux règles du marché, ce qui provoque pour cette discipline le pire et le meilleur.

Pour donner les propriétés des autres catégories de langages, nous nous servirons de la catégorie des langages procéduraux comme référence.

Dans un langage procédural, l'affectation (transfert d'une valeur dans une mémoire) est la base des actions sur les données. La catégorie la plus utilisée après les langages procéduraux est celle des langages fonctionnels.

### *1.2 Les langages fonctionnels*

Dans un langage fonctionnel, les actions reposent sur des fonctions mathématiques ou non qui renvoient des résultats.

Un langage fonctionnel est essentiellement composé d'un dictionnaire de fonctions prédéfinies et d'un mécanisme de construction de nouvelles fonctions par l'utilisateur.

Citons quelques représentants des langages fonctionnels :
**LISP** :(LISt Processing - 1962) en fait c'est essentiellement un langage de traitement de listes.
**SCHEME** : c'est un dialecte pédagogique épuré(1975) de LISP.
**ML** : langage fonctionnel moderne(1990) classé dans la catégorie des langages fonctionnels fortement typés (l'INRIA diffuse gratuitement sur micro-ordinateur une version CAML-Light pour l'enseignement). CAML est utilisé actuellement pour l'enseignement de l'informatique dans les classes préparatoires aux grandes écoles scientifiques françaises.

### *1.3 Les langages logiques*

Citons la catégorie des langages de programmation en logique et son principal représentant :

**PROLOG** (**PRO**grammation en **LOG**ique - 1982).
Dérivé de l'intelligence artificielle, il oblige le programmeur à penser ses actions en termes de buts et à en faire une description relationnelle (vision déclarative).

Le langage Prolog est fondé sur un moteur d'inférence d'ordre 1 (logique des prédicats), et permet l'exploration exhaustive automatique de différents chemins amenant à des solutions. Il possède une qualité intéressante : il est possible d'interpréter un programme prolog d'une manière *déclarative* ou d'une manière *procédurale*.

Le Groupe d'Intelligence Artificielle de Marseille-Luminy fournit des prologs sur micro-ordinateurs à travers la société PrologIA.

### *1.4 Les langages orientés objets (L.O.O)*

Les langages à objets : ils sont fondés sur une seule catégorie d'éléments : " les objets " qui communiquent entre eux grâce à l'envoi de messages (grâce à des opérateurs appelés méthodes). Par rapport à un langage impératif typé, un objet est l'équivalent (mutatis mutandis) d'une variable (simple ou structurée) et la classe dont il est l'instance correspond au type de la variable.

**SIMULA-67** (1967) est le premier langage objet, **SMALLTALK-80**(1980) est un environnement de développement purement objet, **Eiffel**(1990) est un langage objet tourné vers le génie logiciel et la réutilisabilité.

### *1.5 Les langages de spécification*

Les langages de spécification sont encore du domaine de la recherche. Leurs objectifs sont de décrire le plus rigoureusement possible (les modèles principaux sont mathématiques) un logiciel afin de pouvoir le valider et le vérifier.

Nous ne mentionnerons ici que le langage **LPG** de D.Bert(Grenoble) pour les spécifications de types abstraits algébriques, **Z** de J.R. Abrial, le langage dont la notation est fondée sur la théorie des ensembles (puis d'une amélioration de **Z** dénotée **B** par Abrial) et **VDM** langage formel de spécification par pré-condition et post-condition. Ces langages ne peuvent être utilisés d'une manière pratique que sous forme de notation, bien qu'ils soient implantés sur des systèmes informatiques. Ils ne sont pas encore à la disposition du grand public comme les langages des catégories précédentes, bien que certains soient utilisés dans des sites industriels. Par la suite, nous utiliserons un langage de spécification pédagogique fondé sur les types abstraits algébriques.

### *1.6 Les langages hybrides*

Une mention spéciale ici pour des concepts hybrides qui peuvent être de bons compromis entre des catégories différentes. Les concepteurs de tels langages essaient d'importer dans leur langage les qualités inhérentes à au moins deux catégories. La catégorie la plus utilisée est celle des langages impératifs.

Par exemple, la plupart des langages impératifs purs cités plus haut bénéficient d'une " extension " objet, comme **C++** qui est une extension orientée objet du langage **C** conçu à l'origine pour écrire le système d'exploitation Unix.

Plus récemment est apparu un langage comme **Delphi** de Borland qui allie l'approche pédagogique et typée du Pascal, l'approche objet du C++ et les approches visuelles et événementielles de Visual Basic de Microsoft (la sortie fin 2001 de la version entièrement orientée objet de VB, dénommée **VB .Net**, procure à Visual Basic un statut de langage hybride).

Enfin, mentionnons l'important langage **Java** de Sun Microsystems qui permet le développement multi-plateforme en particulier pour l'intranet et qui est grandement utilisé malgré son léger manque de rapidité dû à sa machine virtuelle.

Un mot enfin sur le tout récent langage **C#** support de développement de la plateforme Microsoft .Net, qui a été inventé par le père du langage Delphi (C# s'approprie des avantages de Java et de **Delphi**, il suit de très près la syntaxe de **Java** et celle de **C++**)et qui est le fer de lance de la plateforme .Net de microsoft.

**Object Pascal**, **C++**, **Ada95**, **Java, C#** sont des langages procéduraux qui ont été fortement étendus ou remaniés pour se conformer aux standards objets.

**Remarque de vocabulaire:**

L'ordinateur ne " comprenant " que le langage binaire, il lui faut donc un "traducteur" qui lui traduise en binaire exécutable, les instructions que l'humain lui fournit en langage évolué.

Cette traduction est assurée par un programme appelé **compilateur**.

**Un compilateur du langage L est donc un programme chargé de traduire un programme "source" écrit en L par un humain, en un programme " cible " écrit en binaire exécutable par l'ordinateur.**

# 2.2 : Relations binaires

**Plan du chapitre:**

## 1. Rappel et convention

1. Relation binaire *sur un ensemble*
2. Produit de relations *binaires*
3. Représentation matricielle *d'une relation binaire*
4. *Relation* binaire transposée
5. Matrice du produit *de deux relations*
6. Fermeture transitive *d'une relation binaire*
7. Fermeture réflexo-transitive *d'une relation binaire*
8. Algorithmes *de calcul de matrices*
9. Exemple de calcul *sur une généalogie*

# 1. Rappels et conventions

*Un peu de mathématiques utiles, mais pas trop !*

En informatique, la notion de relation est importante. Nous indiquons ici sans rentrer dans les détails que le lecteur trouvera dans des livres spécialisés, en particulier sur la recherche opérationnelle, comment on implante une relation binaire à travers sa matrice de représentation. Ceci peut donc être considéré comme un bon exemple d'application des matrices booléennes en informatique.

***Convention***

Lorsque nous écrivons " x ← a " ceci se lit: **"x vaut la valeur de a"**.

### *1. Relation binaire sur un ensemble*

Nous appelons relation binaire sur un ensemble E non vide, tout sous-ensemble **R** du produit cartésien E x E.

$$\mathbf{R} \subset E \times E$$

Il est donc possible de définir l'union et l'intersection de deux relations binaires.

### *2. Produit de relations binaires*

Soient $\rho$ et $\sigma$ deux relations binaires sur un ensemble non vide E. On définit le produit des deux relations $\pi = \rho.\sigma$ ainsi :

$$\forall a , a \in E \quad \forall b , b \in E \qquad a \ \rho.\sigma \ b \ \textbf{ssi} \ \exists c , c \in E / (a \ \rho \ c) \text{ et } (c \ \sigma \ b)$$

Nous énonçons brièvement quelques propriétés de ce produit :

- Le produit est associatif.
- Le produit n'est pas commutatif.

**Notations**

| |
|---|
| $\rho^n = \rho.\rho....\rho$ (n fois) |
| $\rho^0$ , est la relation telle que : $\forall a , a \in E$ on a toujours $a \ \rho^0 \ a$ |
| $\rho^{n+m} = \rho^n . \rho^m$ |

### *3. Représentation matricielle d'une relation binaire*

Cas où E est un ensemble fini, c'est d'ailleurs le seul cas qui nous intéresse en informatique où nous ne pouvons pas traiter du non fini.

Soit E l'ensemble : E = { $a_1$, $a_2$, ..., $a_n$ }

- Soit $\rho$ une relation binaire sur E.
- Soit M une matrice carrée d'ordre n sur {0,1}. Nous notons $((m_{i,j}))$ l'élément générique de la matrice M.

Nous dirons que M est la matrice de représentation de la relation binaire $\rho$ et nous la noterons M$\rho$, **ssi** par définition :

**si** $a_i$ $\rho$ $a_j$ **alors** $m_{i,j} \leftarrow 1$ **sinon** $m_{i,j} \leftarrow 0$ **fsi**

*Exemple :*
*E = { 7,8,3 } ;* $\rho$ = { (7,8),(7,3),(3,8),(8,7) }
$a_1 = 7$ ; $a_2 = 8$ ; $a_3 = 3$

Voici la matrice M$\rho$ de la relation $\rho$ définie ci-haut :

$$M\rho = \begin{array}{c} \begin{array}{ccc} 7 & 8 & 3 \end{array} \\ \begin{pmatrix} \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} \end{pmatrix} \begin{array}{c} 7 \\ 8 \\ 3 \end{array} \end{array}$$

### *4. Relation binaire transposée*

- Soit E l'ensemble : E = {$a_1$,$a_2$,...,$a_n$}
- Soit $\rho$ une relation binaire sur E.

Nous notons $\rho^t$ la relation binaire telle que :

$\forall a , a \in E$
$\forall b , b \in E$ $\quad a \ \rho^t \ b$ **ssi** $b \ \rho \ a$

Par construction la matrice de $\rho^t$ est la transposée de la matrice de $\rho$.

$M\rho^t = {}^t M\rho$

### *5. Matrice du produit de deux relations*

En munissant l'ensemble {0,1} d'une structure d'algèbre de boole avec les opérateurs $\wedge$ , $\vee$ , $\neg$ , il nous est possible d'effectuer des calculs sur les matrices de représentation de relations binaires.

- Soient $\rho$ et $\sigma$ deux relations binaires sur un ensemble non vide E. Soit le produit des deux relations , $\pi = \rho.\sigma$, $M\rho$ , $M\sigma$ et $M\pi$ les matrices de $\rho$, $\sigma$ et $\pi$.
- Soit $((a_{i,j}))$ l'élément générique de $M\rho$.
- Soit $((b_{i,j}))$ l'élément générique de $M\sigma$.

La matrice $M\pi = M\rho . \sigma$ est très exactement par définition le produit booléen en croix des matrices $M\rho$ et $M\sigma$.

$$M\rho \times M\sigma = M\pi = M\rho.\sigma = [\bigvee_{k=1}^{n} (a_{i,k} \wedge b_{k,j})]$$

### *6. Fermeture transitive d'une relation binaire*

- Soit E l'ensemble : $E = \{ a_1, a_2, \ldots, a_n \}$
- Soit $\rho$ une relation binaire sur E.

Nous posons par définition sa fermeture transitive qui est la relation binaire $\rho^+$ :

$\rho^+ = \bigcup_{n=1}^{\infty} \rho^n$ , en fait dans le cas où E est fini l'union se limite à un nombre fini **k** de $\rho^n$ distincts donc :

$$\rho^+ = \bigcup_{n=1}^{k} \rho^n$$

En informatique, les ensembles sont toujours finis donc nous considérons que la fermeture transitive de $\rho$ s'écrit :

$$\rho^+ = \rho^1 \cup \rho^2 \cup .... \cup \rho^{k-1} \cup \rho^k$$

### *7. Fermeture réflexo-transitive d'une relationinaire*

- Soit E l'ensemble : E = $\{a_1,a_2,...,a_n\}$
- Soit $\rho$ une relation binaire sur E, sa fermeture transitive $\rho+$ .

On note par définition $\rho*$ sa fermeture réflexo-transitive :

$$\rho^* = \rho^+ \cup \rho^0$$

**Remarque**

Soit un couple (a,b) de E x E tel que a $\rho^*$ b :
a $\rho^*$ b $\Leftrightarrow \exists$ n , n $\in$ E / a $\rho^n$ b

Nous dirons dans ce cas qu'il existe " un chemin de longueur n, allant de a vers b ". En effet d'après la définition du produit :

$$a\ \rho^*\ b \Leftrightarrow \exists (c_1,\ldots,c_n), \forall k, k \in [1,n], c_k \in E$$
tels que : $(a\ \rho\ c_1)$ et $(c_1\ \rho\ c_2)$ ...et $(c_n\ \rho\ b)$

### *8. Algorithmes de calcul de matrices*

**Calcul de la matrice produit à partir de la formule :**

$$M\rho \times M\sigma = M\pi = M\rho.\sigma = [\bigvee_{k=1}^{n} (a_{i,k} \wedge b_{k,j})]$$

Notons $((m_{i,j}))$ l'élément générique de la matrice produit, voici le corps d'un algorithme de calcul de la matrice produit :

```
pour i ← 1 jusquà n faire
 pour j ← 1 jusquà n faire
  S ← 0 ;
  pour k ← 1 jusquà n faire
   S ← S ∨ (a[i,k] ∧ b[k,j])
  Fpour ;
  m[i,j] ← S
 Fpour
Fpour
```

### Algorithme de Warshall pour le calcul de la fermeture transitive :

Avec les mêmes notations de l'algorithme précédent,soit $((a_{i,j}))$ l'élément générique de la matrice Mρ, l'algorithme de Warshall calcule $M\rho^+$ :

```
pour k ← 1 jusquà n faire
 pour i ← 1 jusquà n faire
  pour j ← 1 jusquà n faire
   a_{i,j} ←a_{i,j}∨ (a_{i,k} ∧ a_{k,j})
  Fpour ;
 Fpour
Fpour
```

### *9. Exemple de calcul sur une généalogie*

E = l'ensemble des individus d'une même famille depuis plusieurs générations.

Soient **r**, **s** et **t** les relations binaires :

- x **r** y **ssi** x **est le père de** y
- x **s** y **ssi** x **est la mère de** y
- x **t** y **ssi** x **est un enfant de** y

On peut définir les liens familiaux à l'aide des opérations sur les relations binaires :

$r^2$ = **est grand père paternel de**
$s^2$ = **est grand mère maternelle de**
r.s = **est grand père maternel de**
s.r = **est grand mère paternelle de** (*non commutativité évidente !)*
$r \cup s$ = **est parent de**
$r^n$ = **est arrière arrière...arrière grand père paternel de**
$(r \cup s)^+$ = **est un ancêtre de** (on voit ici la signification pratique de la fermeture transitive qui relie deux individus par un chemin d'ascendants dans son arbre généalogique)
$u.u^t$ = **est frère ou sœur de** etc ....

# 2.3 : Théorie des langages

**Plan du chapitre:**

# 1. Notations et définitions générales

Un langage est fait pour communiquer. Les humains doivent communiquer avec les ordinateurs : ils ont donc élaboré les bases d'une théorie des langages. Dans ce chapitre nous donnons les fondements formalisés d'une telle théorie autour de la notion de grammaire formelle.

**Remarque et convention :**

- Certains éléments d'un langage s'appellent les symboles.
- Soit S un ensemble de symboles ($S \neq \varnothing$). Ce sont les éléments indécomposables dans ce langage (c'est-à-dire non exprimables en autres symboles du langage).

**Définition** *expression sur S*

**On appelle expression sur S, toute suite finie de symboles de S.**

**e** : [ 1, **n** ]→ S , **e** est une expression sur S, **n** est un entier naturel, **n** ≥ 1.

( *e est alors un métasymbole décrivant l'expression S* ).

**Notation**:

On désigne **e** par : **e** = $s_1 s_2 s_3 ... s_n$ , **n** ≥ 1 où : k, $1 \leq k \leq n$ , $s_k \in S$
et par définition e (k) = $s_k$ ( $k \in [1, n]$ ).

On note S+ = { e / ∀e, e expression sur S }
S+ est l'ensemble de toutes les expressions formées sur S.

**Définissons deux opérations sur S+ :**

*L'égalité d'expressions*

Soient **e1** et **e2** deux expressions sur S, on définit leur égalité ainsi :

**e1 = e2** **ssi**

$$\begin{cases} \exists k , k \geq 1 \\ \quad \mathbf{e1} : [1, k] \rightarrow S \\ \quad \mathbf{e2} : [1, k] \rightarrow S \\ \forall i, 1 \leq i \leq k \quad \mathbf{e1}(i) = \mathbf{e2}(i) \end{cases}$$

*la concaténation d'expressions*

soient **e**∈S+ et **f** ∈ S+, on construit le "produit" des deux expressions e.f :

**e** : [1, n] → S
**f** : [1, p] → S
**e.f**: [1, n+p] → S

*avec :*
e.f(i) = e(i) **ssi** i ∈ [1, n]
e.f(i) = f(i) **ssi** i ∈ [n+1, n+p]

**Notation :** (la concaténation de 2 expressions sur S )

Soient **e** et **f** deux expressions :
**e** = $s_1s_2s_3...s_n$
**f** = $t_1t_2t_3...t_p$

**e.f** est notée : $s_1s_2s_3...s_n\ t_1t_2t_3...t_p$

# 2. Grammaire formelle ou algébrique

Comme dans les langages naturels, les informaticiens ont, grâce aux travaux de N.Chomsky, formalisé la notion de grammaire d'un langage informatique.

## *2.1 Monoïde*

**A)** Soit A un ensemble fini appelé alphabet ainsi défini :

$A = \{ a_1, ..., a_n \}$ ( $A \neq \varnothing$ )

**Notations :**

$A^1 = A$
$A^2 = \{ x_1x_2 / (x_1 \in A) \text{ et } (x_2 \in A) \}$
$A^3 = \{ x_1x_2x_3 / (x_1 \in A) \text{ et } (x_2 \in A) \text{ et } (x_3 \in A) \}$
.....
$A^n = \{ x_1x_2....x_n / \ \forall i, 1 \leq i \leq n , (x_i \in A) \}$

**convention**

$A^0 = \{ \varepsilon \}$ (appelé séquence vide)

**B)** On note $A^*$ et $A^+$ les ensembles suivants :

| | |
|---|---|
| $A^* = \bigcup_{n=0}^{\infty} A^n$<br>$A^+ = A^* - \{ \varepsilon \} = A^* - A^0$ | $A^+ = \bigcup_{n=1}^{\infty} A^n$ |

On définit sur $A^*$ une loi de composition interne appelée concaténation, notée $\bullet$ :

$( x, y ) \rightarrow x \bullet y = xy$ *(noms des symboles accolés)*

*La concaténation possède les propriétés suivantes :*

- La loi $\bullet$ est associative :
  $(x \bullet y) \bullet z = x \bullet (y \bullet z)$
- l'élément $\varepsilon$ est un élément neutre pour la loi $\bullet$ :
  $\forall x \in A^*, x \bullet \varepsilon = \varepsilon \bullet x = x$

**Définition :**

$( A^*, \bullet )$ est un monoïde libre

### *2.2 Grammaire formelle*

**Notations** :

Un alphabet est aussi appelé **vocabulaire** ; une **chaîne** ou un **mot** est un élément d'un monoïde ; la **longueur** d'un mot **x** (ou chaîne) est le nombre d'éléments du vocabulaire qui le compose et elle est notée habituellement $|\mathbf{x}|$.

*Exemple* : Vocabulaire V = { a , b }
x = aaabbaab , $x \in V^*$ et $|\mathbf{x}| = 8$

**Remarque :**

*On note* $|\boldsymbol{x}|_a$ le nombre de symboles " a " du vocabulaire V composant le mot **x**.
$x = aaabbaab \Rightarrow |\mathbf{x}|_a = 5$ et $|\mathbf{x}|_b = 3$

**Définition : C-Grammaire**

On appelle **C-Grammaire** (ou, grammaire algébrique de type 2) tout quadruplet :

$G = ( V_N, V_T, S, R )$ où :
$V_N$ est un vocabulaire non terminal ou auxiliaire ( $V_N \neq \varnothing$ )
$V_T$ est un vocabulaire terminal ( $V_T \neq \varnothing$)
$S \in V_N$ , un élément particulier appelé axiome de G
$V_N \cap V_T = \varnothing$
$R \subset ( V_N \cup V_T )^* \times ( V_N \cup V_T )^*$ , R *est un sous-ensemble fini*

**Notations** *:*

- R est appelé l'ensemble des règles de la grammaire G ;
- Une règle $r_i \in R$ est de la forme $( A , \alpha ) / [ A \in V_N$ et $\alpha \in ( V_N \cup V_T )^* ]$ ,
  Elle est notée : $\mathbf{r_i : A \rightarrow \alpha}$
- Lorsque $\alpha \in V_T^*$ , la règle $r_i : A \rightarrow \alpha$ , est dite **règle terminale**.

*Nous ne considérerons par la suite que les grammaires dites de type 2 encore appelées grammaires indépendante du contexte (Context Free), dans lesquelles les règles ont la forme suivante :*

$$\mathbf{R \subset V_N \times ( V_N \cup V_T )^*}$$

### *2.3 Opérations sur les mots*

Soit G une C-Grammaire, $G = (V_N, V_T, S, R)$. On définit sur $( V_N \cup V_T )^*$ une relation binaire appelée " **dérivation directe** " notée $\Rightarrow$ définie comme suit :

**Définition : dérivation directe**

Soient $a \in ( V_N \cup V_T )^*$ et $b \in ( V_N \cup V_T )^*$

On note $a \Rightarrow b$ et l'on dit que **b** dérive directement de **a**, ou que **a** se dérive directement en **b**, si et seulement si

1°) $\exists \alpha \in ( V_N \cup V_T )^*$
2°) $\exists \beta \in ( V_N \cup V_T )^*$
3°) $\exists\ r_i \in R$, telle que : $r_i : A_i \rightarrow \gamma$

4°) a et b s'écrivent :
$a = \alpha A_i \beta$
$b = \alpha \gamma \beta$

**Notation** :

On emploie aussi les termes de " **règle de réécriture** "
ou de " **règle de dérivation** ".

Nous obtenons un processus de construction des mots de la grammaire G par application de la dérivation directe. Si l'on réitère plusieurs fois ce processus de dérivation directe, on obtient à partir d'un mot, une suite de mots de G. En fait il s'agit de construire la fermeture transitive de la relation binaire $\Rightarrow$ . Cette nouvelle relation est appelée la **dérivation dans G** (la dérivation directe en devenant un cas particulier)

**Définition : dérivation**

On dit que **x** se dérive en **y**, s'il existe une suite finie de dérivations directes permettant de passer de **x** à **y** :
(**x**, $x_0$, $x_1$,...,$x_n$ et **y**) étant des mots de $( V_N \cup V_T )^*$on a le chemin suivant :

$$\mathbf{x} \Rightarrow x_0 \Rightarrow x_1 \Rightarrow \ldots x_n \Rightarrow \mathbf{y}$$

on écrit : $x \Rightarrow^* y$ , que l'on lit : **x** se dérive en **y.**

$\Rightarrow^*$ est la fermeture transitive de la relation binaire $\Rightarrow$

### *2.4 Langage engendré par une grammaire*

Nous nous intéressons maintenant à toutes les dérivations possibles construites dans G, par application des règles de G, en privilégiant un point de départ unique pour chacune des dérivations.

Nous avons vu que chaque règle de G commençait en partie gauche par un élément de $V_N$. Nous construisons alors toutes les productions ayant comme point de départ S l'axiome de G. L'ensemble **L** de tous les mots construits s'appelle le " **langage engendré par la grammaire G " :** $L \subset V_T^*$.

**Définition : langage engendré**

Soit la C-grammaire G, G = ( $V_N$ , $V_T$ , S , R )

L'ensemble $L(G) = \{ \mathbf{u} \in V_T^* / S \Rightarrow^* \mathbf{u} \}$
s'appelle le **langage engendré** par G.

*Exemple grammaire $G_0$ :*

$G_0 : V_N = \{ S \}, V_T = \{a,b\}$
**Axiome** : S
**Règles**
1 : $S \rightarrow aSb$
2 : $S \rightarrow ab$

Le langage engendré par $G_0$ est :

**$L(G_0) = \{ a^n b^n / n \geq 1 \}$**

### *2.5 Grammaire d'états finis*

Ce sont des C-Grammaires dans lesquelles les parties droites de règles ont une forme particulièrement simple (on classifie d'ailleurs les grammaires algébriques en général en 4 types en fonction de la forme de leurs règles.

Les C-grammaires sont dites de type-2 et les K-grammaires ou grammaires de Kleene sont dites de type-3).

Pour une grammaire de type-3 ou K-grammaire les règles sont de 2 formes :

**$A \rightarrow a$** **( $a \in V_T$)**
*ou bien*
**$A \rightarrow aB$** **( $B \in V_N$** et **B** pouvant être égal à **A )**

*Exemple* :

$G_1 : V_N = \{ S,A\}$
$V_T = \{a,b\}$
**Axiome** : S
**Règles**
1 : $S \rightarrow a\ S$
2 : $S \rightarrow aA$
3 : $A \rightarrow bA$
4 : $A \rightarrow b$

Le langage engendré par $G_1$ est :

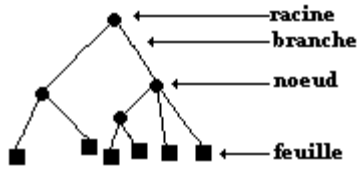
**$L(G_1) = \{ a^n b^p / (n \geq 1)$ et $(p \geq 1) \}$**

### *2.6 Arbre de dérivation d'un mot*

On appelle **arbre** ***A*** toute structure sur un ensemble E qui est :

- soit une structure vide notée ***A***,
- soit une élément noeud **r** associé à un nombre fini d'arbres disjoints vides ou non : $A_1, A_2, \ldots, A_n$ .
- notation : $A = < r, A_1, A_2, \ldots, A_n >$

*Représentation graphique d'un arbre :*




Un arbre est dit " **étiqueté** " si l'on nomme (attribution d'un symbole de nom) sa racine et ses noeuds.

**Définition : arbre de dérivation**

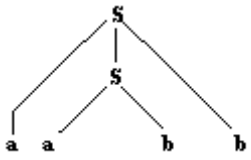
Soit la C-grammaire G, G = ( $V_N$, $V_T$, S, R ).

Un arbre étiqueté est un " **arbre de dérivation** " dans G **ssi** :

- L'alphabet des étiquettes est inclus dans $V_N \cup V_T$.
- Les noeuds sont étiquetés par des éléments de $V_N$.
- Les feuilles sont étiquetées par des éléments de $V_T$.
- L'étiquette de tout noeud est un élément de $V_N$.

Pour tout noeud < **A**, $f_1$, $f_2$, ..., $f_n$> on associe une règle R de la forme : **A** $\rightarrow$ $f_1f_2...f_n$ (règle de dérivation dans G).

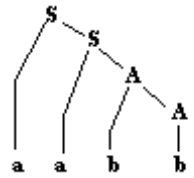
*Exemples :*




*mot $a^2b^2$ dans $G_0$*

Règles de $G_0$ appliquées :

$$S \rightarrow^1 aSb \rightarrow^2 aabb$$




*mot $a^2b^2$ dans $G_1$*

Règles de $G_1$ appliquées :

$$S \rightarrow^1 aS \rightarrow^2 aaA \rightarrow^3 aabA \rightarrow^4 aabb$$

**Définition : grammaire ambiguë**

Une grammaire est dite **ambiguë** si une chaîne a au moins deux arbres de dérivation différents dans G.

*Exemple de grammaire ambiguë :*

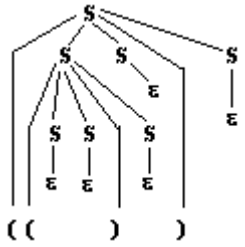
G2 : $V_N$ = {S}
$V_T$ = { ( , ) }
**Axiome : S**
**Règles**
1 : S → (SS)S
2 : S → ε

Le langage engendré par $G_2$ L($G_2$) se dénomme langage des parenthèses bien formées.

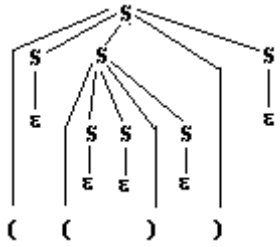
$L(G_2)$ = { ( ) , (( ))( )), ( )( ), ….. }

Soit le mot (( )) de $G_2$ , voici 2 arbres de dérivation de (( )) dans $G_2$ :

Arbre 1 :



Arbre 2 :

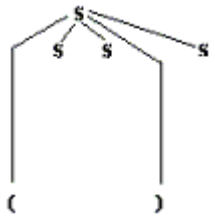


## Arbre 1 correspond dans $G_2$ à la suite des dérivations suivantes :

(afin d'y voir plus clair entre les S choisis nous soulignons les symboles S dérivés)

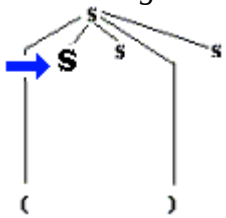
*on part de l'axiome* **S** *et l'on applique la règle 1:*

$S \rightarrow^1 ( SS ) S$



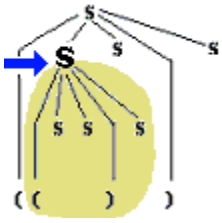
*on applique la règle 1 au premier* ***S*** *de gauche :*

$S \rightarrow^1 ( SS ) S$



*Ce qui donne :*

$S \rightarrow^1 ( \underline{S}S ) S \rightarrow^1 ( ( S S ) S S ) S$



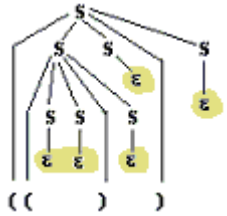
*puis on dérive tous les symboles S à partir de la règle 2 :*

Règle 2

$S \rightarrow \varepsilon$

appliquée 5 fois à :

**( ( S S ) S S ) S**

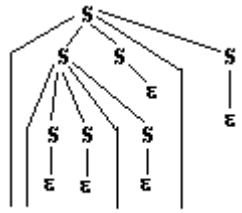


$S \rightarrow \ldots \rightarrow^{2} (( \varepsilon \varepsilon )\varepsilon \varepsilon ) \varepsilon$

Le symbole ε est un élément neutre ($x\varepsilon = \varepsilon x = x$) , nous avons donc comme production finale de cette suite de dérivation le mot : $(( \varepsilon \varepsilon ) \varepsilon \varepsilon ) \varepsilon = (( ))$.

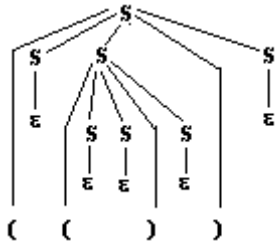
En conclusion, le mot **(( ))** dérive de l'axiome S :

$$S \Rightarrow^{*} (( ))$$



Arbre1 : ( ( ) ) est un arbre de dérivation de mot dans la grammaire $G_2$ .

## Arbre 2 correspond dans $G_2$ à la suite des dérivations suivante :

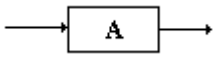
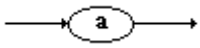
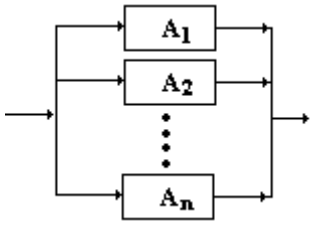
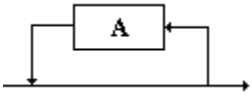
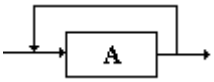


$S \rightarrow^{1} ( S S ) \rightarrow^{1} (S ( S S ) S) \; S \rightarrow^{2} \ldots \rightarrow^{2} (\varepsilon (\varepsilon \varepsilon) \varepsilon) \varepsilon = (( ))$.

Le mot **(( ))** dérive de l'axiome S une seconde fois avec un autre arbre de dérivation distinct du précédent, donc la grammaire $G_2$ est effectivement ambiguë.

### *2.7 Diagrammes syntaxiques*

Il est possible de représenter graphiquement les règles de dérivation d'une grammaire formelle par des diagrammes dénotés " diagrammes syntaxiques ". Cette représentation graphique a pour effet de condenser l'écriture des règles et d'autoriser une meilleure lisibilité.

| REGLES | DIAGRAMMES |
|---|---|
| $\mathbf{A} \in \mathbf{V_N}$ |   |
| $\mathbf{a} \in \mathbf{V_T}$ |   |
| $\mathbf{B \rightarrow \varepsilon}$ |  |
| **B → A1**<br>**B → A2** ou encore :<br>**.....** **B → A1\|...\|An**<br>**B → An** |   |
| **B → AB \| ε**<br>ou :<br>**B → {A}*** |   |
| **B → AB \| A**<br>ou:<br>**B → {A}+** |   |

## 3. Classification de Chomsky des grammaires

Traditionnellement les grammaires algébriques sont classables en quatre catégories qui se différentient par la forme de leurs règles.

Elles sont notées par leur type (type 0, type 1, type 2, type 3). Il existe une relation d'inclusion provenant de leurs définitions :

**type 3 ⊂ type 2 ⊂ type 1 ⊂ type 0**

### *3.1 Les grammaires syntaxiques - type 0*

Les règles ont la forme générale suivante : $\alpha \rightarrow \beta$

pour une règle $\alpha \rightarrow \beta$, les symboles $(\alpha, \beta)$ doivent être de la forme :

$$\alpha \in ( V_N \cup V_T )^+$$
$$\beta \in ( V_N \cup V_T )^*$$

### *3.2 Les grammaires sensibles au contexte - type 1*

Les règles ont la forme suivante : $\alpha A \beta \rightarrow \alpha \gamma \beta$

pour une règle $\alpha A \beta \rightarrow \alpha \gamma \beta$, les symboles$(\alpha, \beta, \gamma$ **,** A**)** doivent être de la forme :

$$A \in V_N$$
$$\alpha \in ( V_N \cup V_T )^*$$
$$\beta \in ( V_N \cup V_T )^*$$
$$\gamma \in ( V_N \cup V_T )^+$$

### *3.3 Les grammaires indépendantes du contexte - type 2*

Les règles ont la forme suivante : $A \rightarrow \alpha$

Pour une règle $A \rightarrow \alpha$, les symboles$(\alpha,A)$ doivent être de la forme :

$$\alpha \in ( V_N \cup V_T )^*$$
$$A \in V_N$$

### *3.4 Les grammaires d'états finis ou de Kleene - type 3*

Les règles n'ont que deux formes possibles :

$A \rightarrow a$ *ou bien* $A \rightarrow aB$

Pour ces règles, les symboles (a, A, B) doivent être de la forme :

$A \in V_N$
$B \in V_N$
$a \in V_T$

# 4. Applications et exemples

### *4.1 Expressions arithmétiques :* une grammaire ambiguë

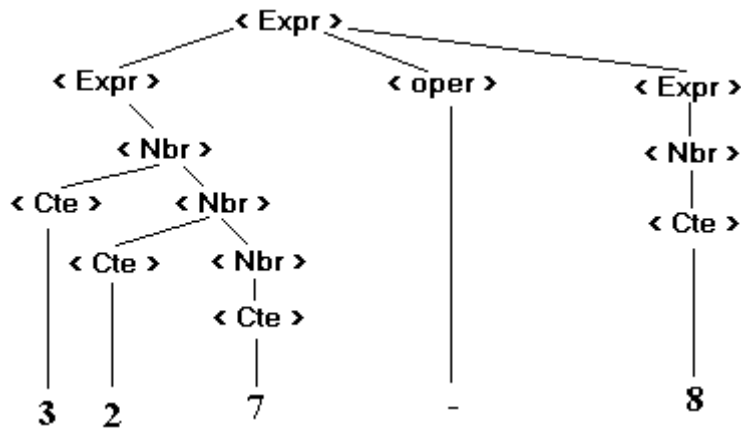
Soit la grammaire $G_{exp}$ = ($V_N$,$V_T$,Axiome,Règles)

**$V_T$** = { **0**, ..., **9**, **+**,**-**, **/**, *****, **)**, **(** }
**$V_N$** = {< Expr >, < Nbr > , < Cte > , < Oper > }
**Axiome** : < Expr >

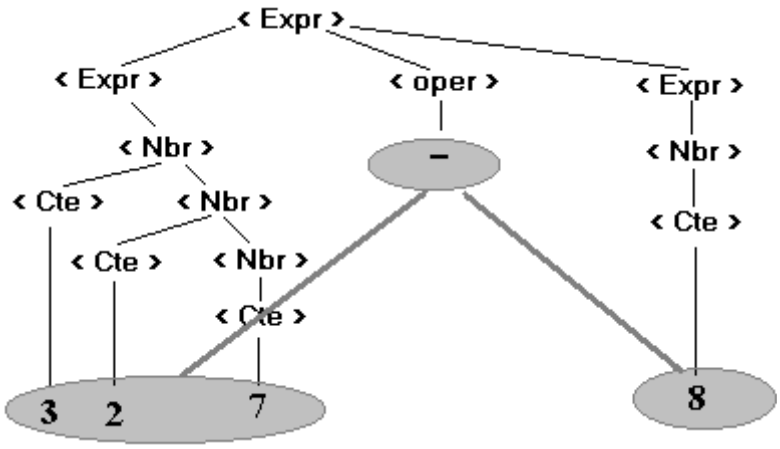
**Règles :**
1 : < Expr > → < Nbr > | **(**< Expr >**)** | < Expr >< Oper >< Expr >
2 : < Nbr > → < Cte > | < Cte >< Nbr >
3 : < Cte > → **0** | **1** |...| **9**
4 : < Oper > → **+** | **-** | ***** | **/**

Les mots de L($G_{exp}$) sont des expressions de la forme (x+y-z)*x etc... où x, y, z sont des entiers.

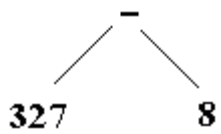
*Exemple* : **327 - 8** est un mot de L($G_{exp}$)
Ce mot n'a qu'un seul arbre de dérivation dans $G_{exp}$, dessinons son arbre de dérivation :



Nous pouvons faire ressortir les liens abstraits qui relient les trois éléments terminaux 327, - et 8 :



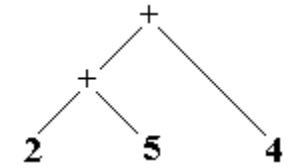

L'arbre obtenu en grisé à partir de l'arbre de dérivation s'appelle l'arbre abstrait du mot " 327-8 ".



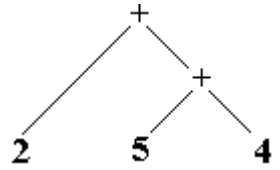

Cet arbre abstrait permet de manipuler la structure générale du mot facilement tout en résumant la structure générale de l'arbre de dérivation.

Soient trois autres mots de $L(G_{exp})$ **2+5+4**, **2-5+4** et **2+5*4**, ils ont chacun deux arbres de dérivation. Nous donnons ci-après deux arbres abstraits de chaque mot.

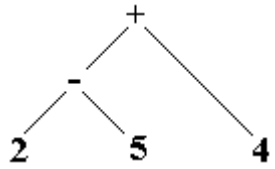
Arbre-1 : 2+5+4



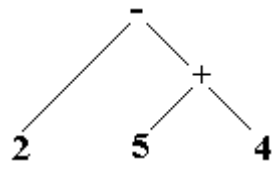

Arbre-2 : 2+5+4



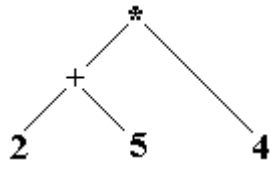

Arbre-3 : 2-5+4



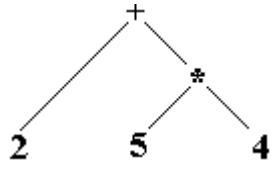

Arbre-4 : 2-5+4




Arbre-5 : 2+5*4




Arbre-6 : 2+5*4




Nous remarquons donc que $G_{exp}$ est une grammaire ambiguë puisqu'il existe un mot possédant au moins deux arbres de dérivation.

Pour l'instant les mots **2+5+4**, **2-5+4** et **2+5*4** ne sont que des concaténations de symboles sans aucun sens particulier

Si nous voulions aller plus loin en donnant un sens (de la **sémantique**) à ces mots de telle façon qu'ils représentent des calculs sur les entiers avec les propriétés classiques des opérations sur les entiers, nous pourrions nous trouver un " *bon choix* " parmi les arbres abstraits précédents.

Nous appellerons ces choix " interpréter " l'expression.

**Examen de la situation pour le mot 2+5+4 :**

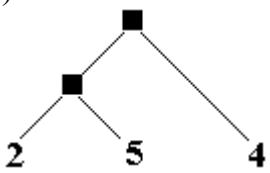
- Arbre-1 s'interprète : (2+5)+4
- Arbre-2 s'interprète : 2+(5+4)

L'opérateur " + " est associatif donc pour notre interprétation les deux arbres 1 et 2 peuvent convenir.

**Examen de la situation pour le mot 2-5+4 :**

- Arbre-3 s'interprète : (2-5)+4
- Arbre-4 s'interprète : 2-(5+4)

Les opérateurs + et - sont de même priorité et nous obtenons deux expressions différentes selon le choix de l'arbre.
Traditionnellement lorsque deux opérateurs ont la même priorité, l'évaluation se fait à partir de la gauche de l'expression. Donc l'arbre 3 conviendrait.
Nous pourrions penser lever l'ambiguïté en choisissant systématiquement l'arbre abstrait d'évaluation à gauche correspondant à un parenthésage implicite à gauche(comme arbre-1 et arbre-3) :




Nous allons voir ci-dessous que ce n'est pas possible.

**Examen de la situation pour le mot 2+5*4 :**

- Arbre-5 s'interprète :(2+5)*4
- Arbre-6 s'interprète : 2+(5*4)

Les opérateurs + et * n'ont pas la même priorité. Nous obtenons deux expressions différentes selon le choix de l'arbre. Mais ici c'est le choix de l'arbre 6 qui s'impose à cause de la priorité du * sur le +.

Nous avons fait ressortir le fait qu'il était impossible de privilégier systématiquement pour " l'interprétation " des expressions une catégorie d'arbre plutôt qu'une autre, il faut donc changer de grammaire et éviter l'ambiguïté.

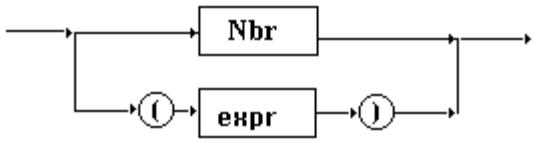
### *4.2 Expressions arithmétiques* : une grammaire non ambiguë

Nous donnons ci-dessous une grammaire non ambiguë basée sur la précédente et tenant compte de la précédence (priorité d'opérateur). Nous séparons les opérateurs en deux catégories ; les opérateurs de priorité zéro (Oper_0) et ceux de priorité un (Oper_1).

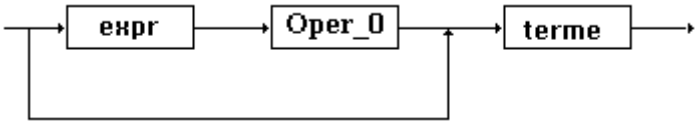
$V_T$ = { **0**, ..., **9**, **+**,**-**, **/**, *****, **)**, **(** }
$V_N$ = {< Expr >, < Nbr > , < Cte > , < Oper_0 > , < Oper_1 >, < facteur >, < terme > }
**Axiome** : < Expr >

**Règles :**

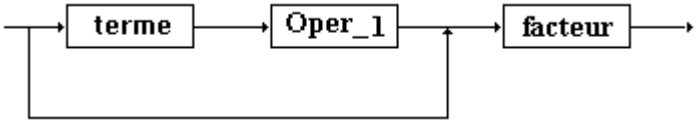
**facteur**



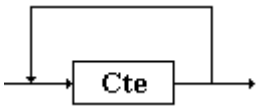

**expr**



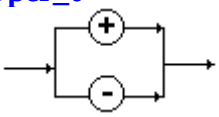

**terme**



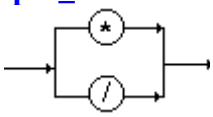

**Nbr**



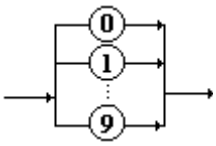

**Oper_0**




**Oper_1**




**Cte**




En pratique ce ne sera pas une telle de grammaire qui sera retenue pour le calcul des expressions arithmétiques car elle contient une règle récursive gauche, ce qui la rend difficilement analysable par des procédés simples.

# 2.4 : Une grammaire du Pascal

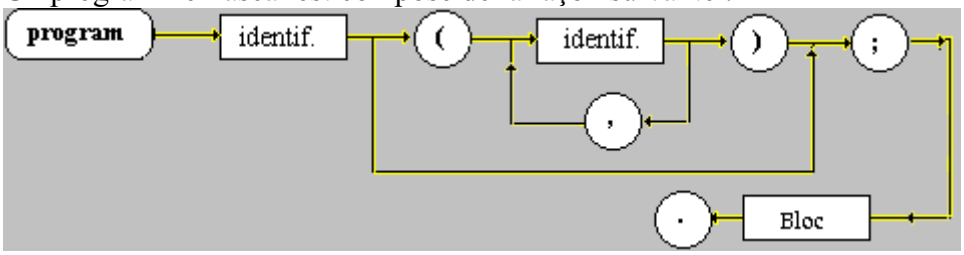
**Plan du chapitre:**

# 1. Rappel de la structure d'un programme Pascal

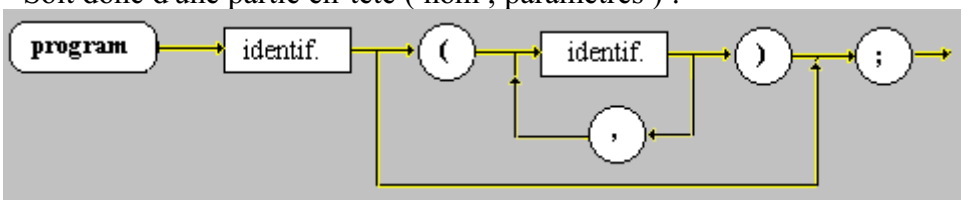
*Il ne s'agit pas d'apprendre le langage pascal, mais plutôt d'un résumé visant à se remémorer les principes de base du langage, et ainsi de se familiariser avec les principes utiles et pratiques du langage relativement à la programmation. La société Borland-Inprise met sur son site web, gratuitement par téléchargement, des compilateurs pascal anciens mais efficaces pour le débutant* ***(http://www.borland.fr)****.*

Nous utilisons une description d'un Pascal-Delphi réduit à l'aide des diagrammes syntaxiques.

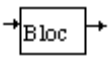
Un programme Pascal est composé de la façon suivante :




- Soit donc d'une partie en-tête ( nom , paramètres ) :




- d'une partie corps (ou Bloc) :



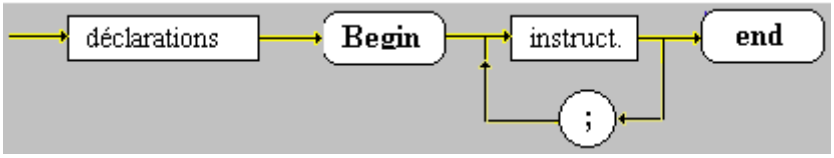

- et se termine par un point :

*Exemples d'en-tête :*

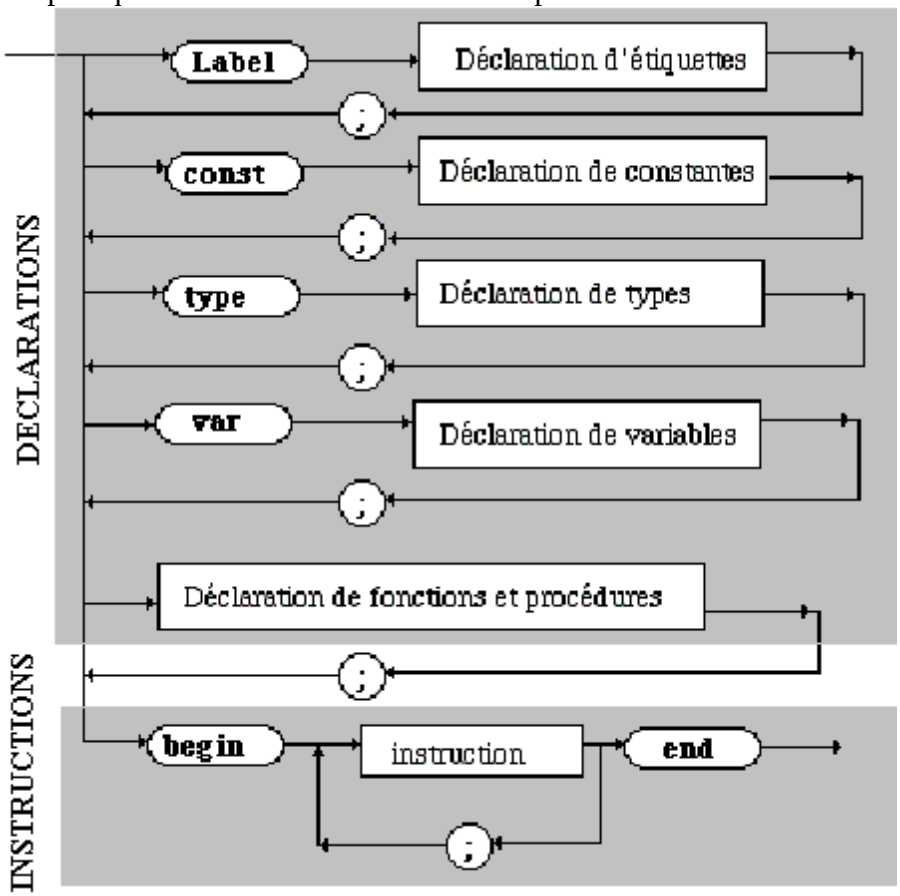
1°) **program** exemple_01 ( input, output ) ;
2°) **program** exemple_02 ;

Le langage Pascal étant structuré, un bloc est composé de sections ou paragraphes bien séparées :

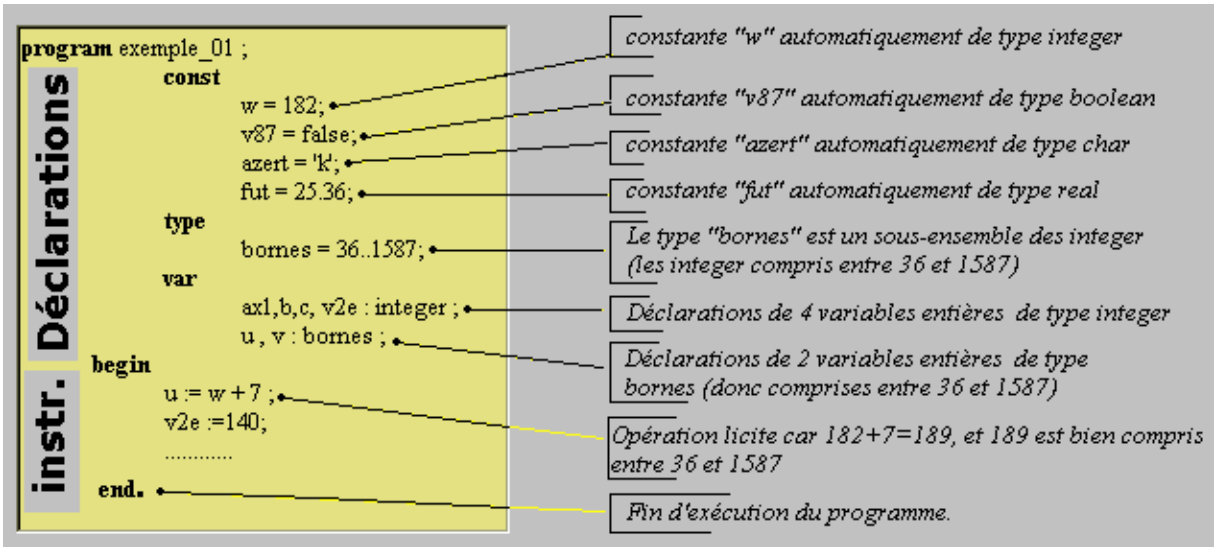
**Bloc :**



En pratique un bloc Pascal contient deux parties : des déclarations et des instructions




Exemple de programme *avec Bloc :*



## 2. Les opérateurs en pascal

Ce sont les règles de composition qui précisent la priorité retenue entre les différents opérateurs du langage. Ces priorités sont réparties en **4** niveaux :

- plus haut niveau de priorité 4 : opérateur unaire **not**
- niveau de priorité 3 : opérateurs multiplicatifs (***, /, div, mod, and** )
- niveau de priorité 2 : opérateurs additifs ( **+ , - , or** )
- plus bas niveau de priorité 1 : opérateurs relationnels (< , >, etc…)

### *2.1 Liste de tous les opérateurs selon le type de données en Pascal.*

**integer x integer ⟶ integer**

| op | signification | exemple |
|---|---|---|
| * | multiplication | 2 * (x-8)+a*b |
| div | division euclidienne | u div (x*8)+a div b |
| + | addition | x - (x+8)+a*b |
| - | soustraction | x - (x-8)-a-b |
| mod | reste euclidien | |

*opérateurs sur les entiers*

**real x real ⟶ real**

| op | signification | exemple |
|---|---|---|
| * | multiplication | 2 * (x-8)+a*b |
| / | division | u / (x*8)+a/b |
| + | addition | x - (x+8)+a*b |
| - | soustraction | x - (x-8)-a-b |

*opérateurs sur les réels*

**bolean x boolean ⟶ boolean**

| op | signification | exemple |
|---|---|---|
| or | ou | (a or b) or ((c or d)) |
| and | et | (a and b) and ((c and a or d)) |
| not | non | not a or not( b and c) |

*opérateurs sur les booléens*

**set of x set of → set of**

| op | signification | exemple |
|---|---|---|
| * | intersection | A * B |
| + | union | A+B |
| - | différence | A-B |

**T x set of → boolean**

| op | signification | exemple |
|---|---|---|
| in | appartient à | x in E |

**set of x set of → boolean**

| op | signification | exemple |
|---|---|---|
| = | égalité | A =B |
| <> | différent de | A<>B |
| <= | inclusion | A <= B |

*opérateurs sur les ensembles : set of*

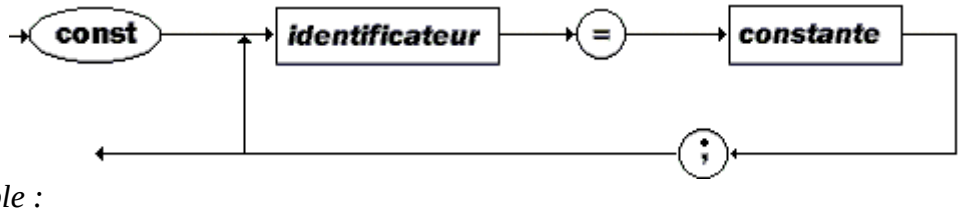
**T x T ⟶ boolean**

| op | signification | exemple |
|---|---|---|
| < | inférieur strict | 'b' < 'k' ; -8 < 25 |
| > | supérieur strict | 'p' > 'k' ; 98 > 32 |
| = | égalité | x = 3 ; a = b |
| <= | inférieur large | 'b' <= 'k' ; -8 <= 25 |
| >= | supérieur large | 'p' >= 'k' ; 98 >= 32 |
| <> | différent | 'g'<>'m' ; 0 <>x |

*opérateurs de comparaison sur un type T*

### *2.4 Déclarations des constantes*

Sert à associer un identificateur à une valeur de constante, sa valeur est non modifiable dans le reste du programme. Il existe 3 identificateurs de constantes prédéfinis : **True** , **False** , et **Nil** .




*Exemple :*

```
program exemple_03 ;
const
   x = 12; <---------- x est une constante de type integer.
   a2 = true; <---------- a2 est une constante de type boolean.
   Y = 'h'; <---------- Y est une constante de type char.
   r2 = 25.36; <---------- r2 est une constante de type real.
```

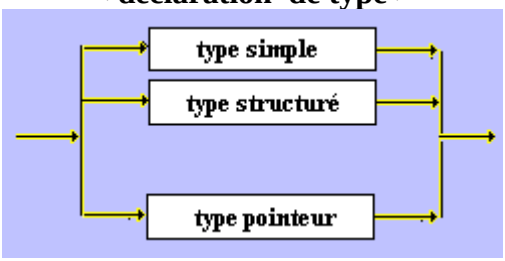
# 3. Déclarations des types en pascal

Les types sont utilisés pour créer de nouveaux domaines de définition de variables. Une déclaration d'un nouveau type de données sert à associer un identificateur à un type de données construit par l'utilisateur.

Cette construction est élaborée à l'aide de constructeurs de type et détermine l'ensemble des valeurs possibles des variables du nouveau type.

On classe les types en 3 catégories :

**< déclaration de type >**

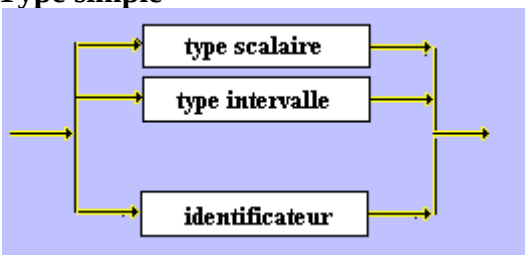



### *3.1 Déclarations des types simples*

Cette déclaration est composée des :

- type scalaire
- type intervalle
- identificateur d'un type déjà déclaré

**< Type simple >**




**< Les types scalaires > ( ils sont de 2 sortes ) :**

Les types prédéfinis :

- **integer**
- **real**
- **char**
- **boolean**
- **string**

Les types énumérés :

identif0 = **(** identif1,identif2,....,identifk **)**

### *3.2 Déclarations des types énumérés*

**Type** identif0 = **(** identif1,identif2,....,identifk **) ;**

Il s'agit ici d'une définition en extension des éléments du type. Les *identifn* sont des constantes symboliques de base du type et doivent être tous différents dans la même énumération, et ne peuvent se retrouver ni dans une autre énumération, ni redéfinis ailleurs.

Ce type est doté d'une fonction spécifique : **ord** qui dénote le numéro d'ordre d'un élément dans l'ensemble des valeurs du type (attention l'ordre est construit de gauche à droite et la numérotation débute à la valeur 0).

*Exemple :* **créons un type jour de la semaine**

```
Type
   jour = ( lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche ) ;
```

Le type énuméré jour voit ses constantes de base automatiquement numérotées de 0 à 6 comme l'indique le tableau ci-après:

| lundi | mardi | mercredi | jeudi | vendredi | samedi | dimanche |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

*Ainsi le rang est accédé par la fonction **ord** :*

**ord**(jeudi) = 3
**ord**( lundi) = 0

**Remarque** *:*

Les types scalaires sauf le type real bénéficient de 2 fonctions **succ** et **pred**

**succ** : T $\rightarrow$ T / **succ** ($a_i$) = $a_{i+1}$ (successeur dans T , lorsqu'il existe)
**pred** : T $\rightarrow$ T / **pred** ($a_i$) = $a_{i-1}$ (prédécesseur dans T, lorsqu'il existe )

### *3.3 Déclarations des types intervalles*


Il peut être défini comme un intervalle fermé borné d'un autre type scalaire, sauf **real**. Les constantes représentent les bornes de l'intervalles (la constante de gauche représente la borne inférieure, la constante de droite représente la borne supérieure)

*Exemples :*

```
Type
   jour = (lundi , mardi , mercredi , jeudi , vendredi , samedi , dimanche ) ;

   mois = 1..12 ;  // intervalle sur les entiers compris entre 1 et 12
   week_end = vendredi..dimanche ; // intervalle sur les jour : vendredi , samedi , dimanche
   lettre_min = 'a'..'z' ; // intervalle sur les caractères de type lettres minuscules
   lettre_maj = 'A'..'Z' ; // intervalle sur les caractères de type lettres majuscules
```

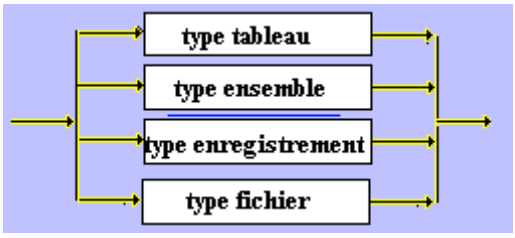
Il est bien entendu possible de déclarer ensuite des variables sur ces types :

```
   Var
      x : mois ;
      y : week_end
      z : lettre_maj
```

### *3.5 Déclarations des types structurés*

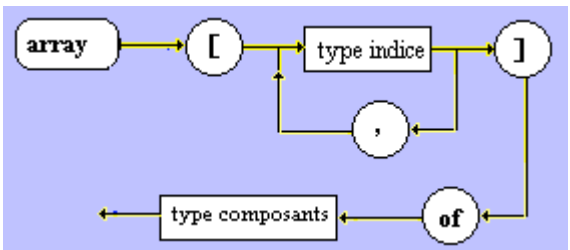
Il est donc possible en Pascal de construire et d'utiliser des variables de type simple comme integer, real, boolean, string, char, énumérés et intervalles. Mais il est aussi possible de travailler avec des familles de variables de même type ayant une structuration spécifique ou bien avec des structures contenant des variables ayant des types différents. Ces familles sont appelées des types structurés.

Elles sont au nombre de 4 en pascal :




Une définition de type structuré, précise par l'intermédiaire du constructeur de type, la méthode de structuration et le type des données le composant.

### *3.6 Déclarations de type tableau*



Le type tableau est défini par le constructeur de type **array[ ] of.**

C'est une structure homogène, formée d'un nombre fixe de composants qui sont tous du même type de base. Tous les composants d'un tableau sont désignés par des indices, qui sont des expressions appartenant au **type indice** du tableau.

Un tableau est en fait une structure de donnée à *accès aléatoire* , c'est à dire que tous ses composants peuvent être sélectionnés et atteints de manière égale. Ils sont rangés dans l'ordre des indices.

Un tableau à n dimensions (un vecteur est représenté par un tableau à une dimension, une matrice par un tableau à deux dimensions...) est défini par n **types d'indices** séparés par des virgules.

Un **type indice** est un type simple sauf **real** et **integer**.

*Exemple :*

```
Type
  jour = ( lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche ) ;
  mois = 1..12 ;
  week_end = vendredi..dimanche ;
  lettre_min = 'a'..'z' ;
  lettre_maj = 'A'..'Z' ;
  tableau_01 = array[jour] of mois;
  tableau_02 = array[jour] of array[1..30] of mois;
  tableau_03 = array[jour,1..30] of mois;
  tableau_04 = array[lettr_min,0..5,jour,boolean] of char;
var
  T1 : tableau_01;
  T2 : tableau_02;
  T3 : tableau_03;
  T4 : tableau_04;
  etc.....
```

**ATTENTION :**

*Notons ici que malgré la similitude de construction des deux types tableau_02 et tableau_03 (ce sont des types de matrices où l'indice ligne varie dans le type jour, et l'indice colonne varie dans le type 1..30), ce ne sont pas des types identiques, car ils sont déclarés séparément.*

Donc dans l'exemple précédent, T2 et T3 **ne** sont **pas** des tableaux du même type.

## Accès aux variables d'un type tableau

Il faut, afin de pouvoir accéder à un composant d'un tableau, utiliser des indices obligatoirement de même type et en même nombre qu'indiqués dans la déclaration.

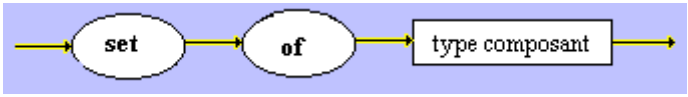
*Exemple : (en reprenant les déclarations précédentes*)

```
var
   T1 : tableau_01;
   T2 : tableau_02;
   T3 : tableau_03;
   T4 : tableau_04;
   m : mois;
   j : jour;
   k : 1..30;
   L,b: boolean;
   n : integer;
   c : lettre_min;
```

## Les écritures suivantes sont licites :

```
 j:= jeudi; k:= 20; c:='f'; L:=false; b:=true; n:=2;
T1[mardi]:= 8; T1[j]:= 10;
T2[mardi,5]:= 8; T2[mardi] [5]:= 8; T2[j,k-3]:= 8; T2[j] [k-3]:= 8;
T3[mardi,5]:= 8; T3[mardi] [5]:= 8; T3[j,k]:= 8; T3[j] [k]:= 8;
T4['t',3,samedi,true]:= 'h'; T4['t'][3][samedi][true]:= 'h';
T4[c,n+2,j,L or b]:= '+'; ..... etc
```

### *3.7 Déclarations de type ensemble*




Un type ensemble est défini d'un manière extensive par le constructeur **set of**, le domaine des valeurs de ses éléments par son type de base.

le t**ype ensemble** est un type simple sauf **real** et **integer**.

C'est un ensemble fini et l'on peut construire tous ses sous-ensembles :

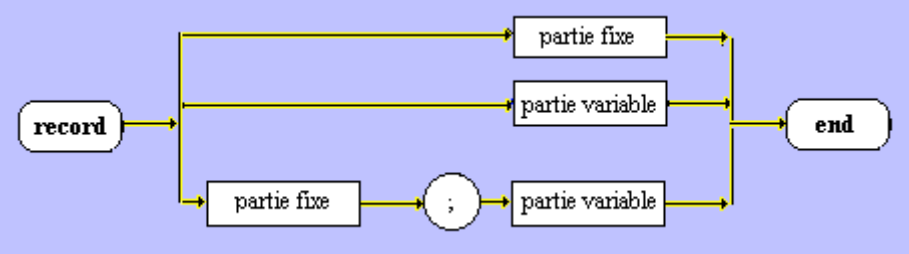
*Exemple :*

```
Type
   couleur = (noir,blanc);
   ens_couleur = set of couleur;
var
   x,y,z,t : ens_couleur;
```

```
begin
   x := [ ] ; <--- ensemble vide (0 élément)
   y := [noir]; <--- ensemble (1 élément)
   z := [blanc]; <--- ensemble (1 élément)
   t := [noir,blanc]; <--- ensemble (2 éléments : maximum possible de l'exemple)
etc.....
```

On peut dire en fait que le type ens_couleur est l'ensemble P(couleur) (ensemble des parties) et que toute variable du type ens_couleur est un sous-ensemble de P(couleur).

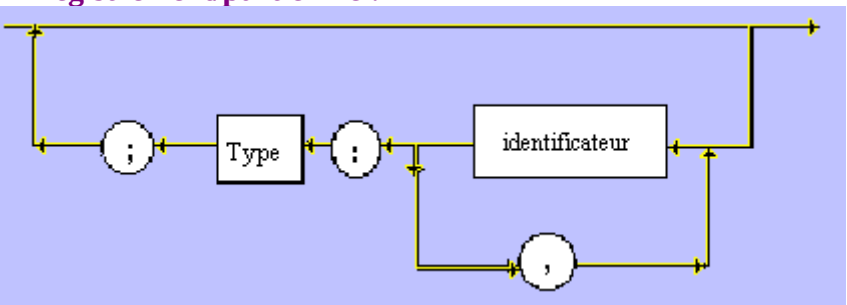
### *3.7 Déclarations de type enregistrement*




Le type enregistrement est une collection de composants appelés **champs** de l'enregistrement. Ils peuvent être d'un type quelconque sauf le type fichier. C'est une structure hétérogène.

> Tous les identificateurs de champs d'une même structure enregistrement doivent être différents à l'intérieur de l'enregistrement. Ils permettent d'accéder directement aux éléments de l'enregistrement.

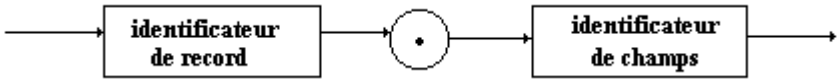
**Enregistrement/partie fixe :**




*Exemple:*

```
Type
   enregis = record
      jour : (lundi,mardi,dimanche);
      x,y : integer;
      mois : 1..12;
      T_paie : array[boolean,1..30] of real;
   end;
```

**Enregistrement/accès aux champs :**




L'accès aux champs à l'intérieur d'un enregistrement s'effectue à l'aide de l'identificateur de l'enregistrement (**identif de record**), puis de celui du champs (**identif de champs**) auquel on désire accéder, dans cet ordre, comme en désignant un chemin accédant aux éléments en écrivant de gauche à droite.

*Exemple :*

```
Type
  Tenregis = record
      jour : (lundi,mardi,dimanche);
      x,y : integer;
      mois : 1..12;
      T_paie : array[boolean,1..31] of real;
  end;
var
  A : Tenregis;
begin
  A.jour:=;mardi;
  A.mois:=8;
  A.y:=125;
  A.x:=0;
  A.T_paie[false,A.mois] := -2.37
  etc.....
```

## 4. Instructions en pascal

Ce sont les traductions des instructions algorithmiques de notre langage de description formelle d'algorithme que nous avons dénommé LDFA.

| LDFA | Pascal |
|---|---|
| Ω (instruction vide) | pas de traduction |
| **debut** i1 ; i2; i3; ..... ; ik **fin** | **begin** i1 ; i2; i3; ..... ; ik **end** |
| x ← a | x := a |
| ; | (ordre d'exécution) ; |
| **Si** P **alors** E1 **sinon** E2 **Fsi** | **if** P **then** E1 **else** E2<br>*( attention défaut, pas de fermeture !)* |
| **Tantque** P **faire** E **Ftant** | **while** P **do** E<br>*( attention, pas de fermeture)* |

| | |
|---|---|
| **répeter** E **jusquà** P | **repeat** E **until** P |
| **lire** (x1,x2,x3.....,xn ) | read(fichier,x1,x2,x3.....,xn )<br>readln(x1,x2,x3.....,xn )<br>Get(fichier) |
| **ecrire** (x1,x2,x3.....,xn ) | write(fichier,x1,x2,x3.....,xn )<br>writeln(x1,x2,x3.....,xn )<br>Put(fichier) |
| **pour** x ← a **jusquà** b **faire** E **Fpour** | **for** x:=a **to** b **do** E *(croissant)*<br><br>**for** x:=a **downto** b **do** E *(décroissant)*<br>*( attention, pas de fermeture)* |
| **SortirSi** P | **if** P **then** Break |

### *4.1 Instruction d'affectation*

L'affectation est applicable à tous les genres de variables du pascal sauf au type **file of**.


**Sémantique:**

- Evaluation de la partie droite (l'expression)
- Transfert de la valeur calculée dans la partie gauche (la variable)

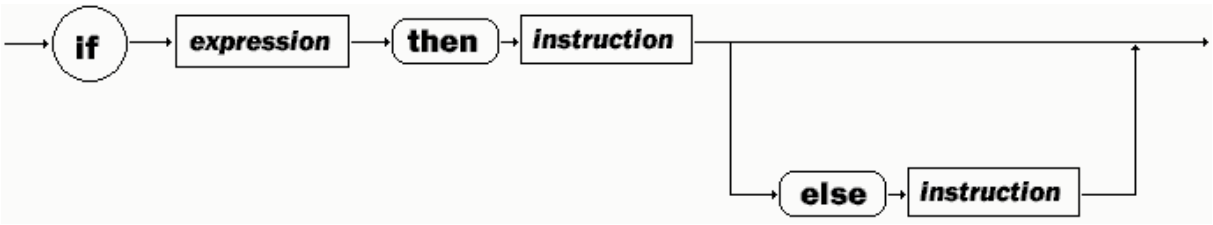
*Exemple :*

```
program Affectation ;
type
   Temperature = -20 .. 40 ;
   LettreMin = ' a ' .. ' z ' ;
   Jour = ( lundi , mardi , mercredi , jeudi ) ;
var
    a : integer ;  b : char ;
    c : string ;
    Temp : Temperature ; Lmin :  LettreMin ;
    Day : Jour ;
begin
   Temp := 18 ;
    a := (2+Temp)*4 ;
    b := 'F' ;
    c := 'bon'+'jour' ;
    Lmin := 'f' ;
    Day := mercredi ;
end.
```

Après affectations :

**Temp** *vaut* 18
**a** *vaut* 80
**b** *vaut* 'F'
**c** *vaut* 'bonjour'
**Lmin** *vaut* 'f'
**Day** *vaut* mercredi

### *4.2 Instruction de condition*




Dans l'instruction **if**, l'expression est un prédicat ( expression contenant des variables, prenant la valeur vrai ou faux), les blocs <***instruction***> représentent soit une instruction simple, soit une instruction composée (**begin** ..... **end**).

**Sémantique:**

cas du ***if...then***

- Si l'expression est vraie, le bloc d'instruction situé après le **then** est exécuté et le ***if...then*** s'arrête
- Si l'expression est fausse le ***if...then*** s'arrête.

cas du ***if...then...else***

- Si l'expression est vraie, le bloc d'instruction situé après le **then** est exécuté et le ***if...then...else*** s'arrête.
- Si l'expression est fausse, le bloc d'instruction situé après le **else** est exécuté et le ***if...then...else*** s'arrête.

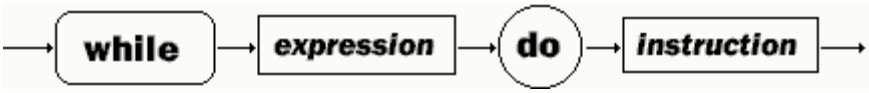
*Exemple :*

```
program Condition ;
var
    x, y ,z : integer ;

begin
   x := 10 ;
   y := x*4 ;
   if y>100 then z := y
   else z := 0;
   if z = 0 then
      y := 0
   x := 0 ;
end.
```

*Exécution pas à pas :*

**x** *vaut* 10
**y** *vaut* 40
**y>100** *est* **false**
donc **z** *vaut* 0
**z=100** *est* **true**
**donc y** *vaut* 0
**x** *vaut* 0
*(à la fin : x=0, y=0, z=0)*

### *4.3 Instruction d'itération while…do*




Dans l'instruction **while…do**, l'expression est un prédicat ( expression contenant des variables, prenant la valeur vrai ou faux), le blocs <***instruction***> représente soit une instruction simple, soit une instruction composée (**begin** ..... **end**).

**Sémantique:**

C'est une instruction de boucle.

- Tant que l'expression reste vraie, le bloc d'instruction est réexécuté.
- Dès que l'expression est fausse le ***while…do*** s'arrête.

C'est une boucle non finie (c-à-dire que l'on ne peut pas connaître dans les cas de figure si une boucle quelconque de ce type s'arrêtera après un nombre fini d'exécution).

*Exemple :*

```
program WhileDo ;
var
    x, y  : integer ;

begin
   x := 1 ;
   y := 0 ;
   while x<4 do
   begin
     x := x+1 ;
     y := y +x
   end;
   writeln ('x=', x , 'y=', y)
end.
```

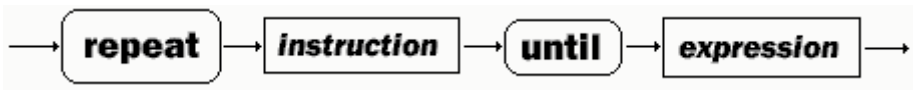
*Le programme écrit :*

x=4 y=9

*Exécution pas à pas :*

**x** *vaut* 1
**y** *vaut* 0
**x<4** *est* **true**
  donc **x** *vaut* x+1 *soit* 2
  et **y** *vaut* y+x soit 2
**x<4** *est* **true**
  donc **x** *vaut* x+1 *soit* 3
  et **y** *vaut* y+x *soit* 5
**x<4** *est* **true**
  donc **x** *vaut* x+1 *soit* 4
  et **y** *vaut* y+x *soit* 9
**x<4** *est* **false**  donc **arrêt**
*(à la fin : x=4, y=9)*

### *4.4 Instruction d'itération repeat…until*



Dans l'instruction **repeat…until**, l'expression est un prédicat ( expression contenant des variables, prenant la valeur vrai ou faux), le blocs <***instruction***> représente soit une suite d'instructions simples.

**Sémantique:**

C'est une instruction de boucle.

- Tant que l'expression reste fausse, le bloc d'instruction est réexécuté.
- Dès que l'expression est vraie le ***repeat…until*** s'arrête.

C'est une boucle non finie (c-à-dire que l'on ne peut pas connaître dans les cas de figure si une boucle quelconque de ce type s'arrêtera après un nombre fini d'exécution).

La différence avec le **while .. do** réside dans le fait que le **repeat ... until** exécute toujours au moins une fois le bloc d'instructions avant d'évaluer l'expression booléenne alors que le **while ... do** évalue immédiatement son expression booléenne avant d'exécuter le bloc d'instructions.

*Exemple :*

```
program RepeatUntil ;
var
    x, y  : integer ;

begin
   x := 1 ;
   y := 0 ;
   repeat
     x := x+1 ;
     y := y +x
   until x>=4;
   writeln ('x=', x , 'y=', y)
end.
```

*Le programme écrit :*

```
x=4 y=9
```

*Exécution pas à pas :*

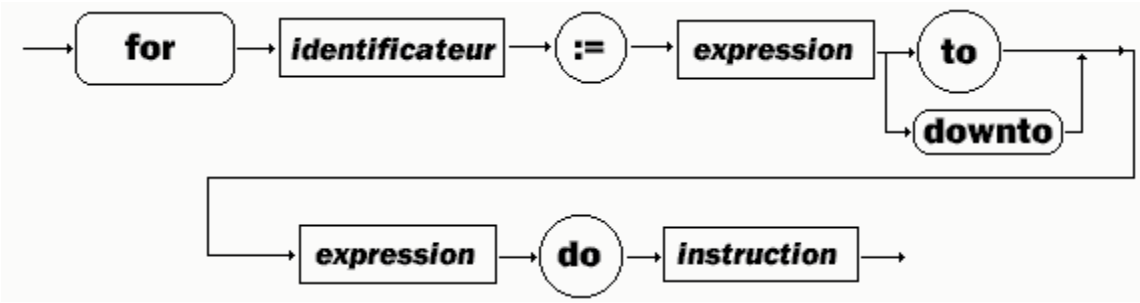
**x** *vaut* 1
**y** *vaut* 0
***on entre dans le repeat***
donc **x** *vaut* x+1 *soit* 2
et **y** *vaut* y+x soit 2
**x>=4** *est* **false**
donc **x** *vaut* x+1 *soit* 3
et **y** *vaut* y+x *soit* 5
**x>=4** *est* **false**
donc **x** *vaut* x+1 *soit* 4
et **y** *vaut* y+x *soit* 9
**x>=4** *est* **true** donc **arrêt**
*(à la fin : x=4, y=9)*

Ce programme fournit le même résultat que celui de la boucle **while…do**, car il y a une correspondance sémantique entre ces deux boucles :

| | |
|---|---|
| **repeat** <instruction> **until** <expr> | <instruction> **;**<br>**while not**<expr> **do** <instruction> |

### *4.5 Instruction d'itération for…do*

C'est une instruction de boucle, il y a deux genres d'instructions **for (for...to** et **for...downto) :**




Version **for** <***identificateur***> := <***Expr1***>**to** <***Expr2***> **do** <***Instruction***> :

- ***identificateur*** est une variable qui se dénomme indice de boucle.
- <***Expr1***> et <***Expr2***> sont obligatoirement des expressions du même type que la variable d'indice de boucle ***identificateur***.
- < ***Instruction*** > est un bloc d'instruction simple ou composée (**begin** ..... **end**).

Version **for** <***identificateur***> := <***Expr1***> **downto** <***Expr2***> **do** <***Instruction***> :

- même signification des constituants que pour la version précédente, seul le sens de parcours différe (par valeurs croissantes pour un **for...to**, par valeurs décroissantes pour un **for...downto**).

**Sémantique:**

L'indice de boucle prend toutes les valeurs (par ordre croissant ou décroissant selon le genre de **for**) comprises entre <***Expr1***> et <***Expr2***> bornes inclues.

Tant que la valeur de l'indice de boucle ne dépasse pas
- par valeur supérieure dans le cas du **for...to**,
- ou par valeur inférieure dans le cas du **for...downto**

la valeur de <***Expr2***>, le bloc d'instruction est réexécuté.

C'est une boucle **finie** (c-à-dire que l'on connaît à l'avance le nombre de tours de boucle).

*Exemple :*

```
program ForDo ;
var   x, y  : integer ;
   begin
     y :=0 ;
     for x := 1 to 3 do
          y := y +x
   end.
```

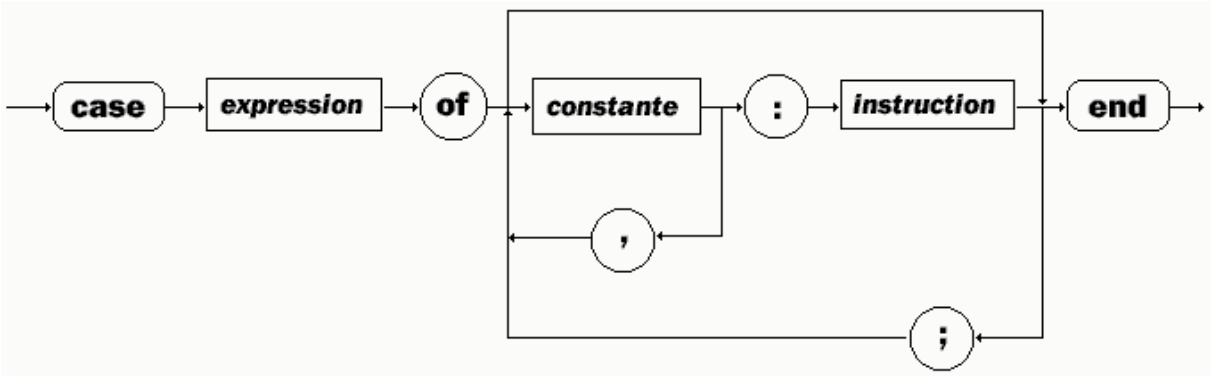
*Exécution de chaque tour de boucle :*

**y** *vaut* 0
**x** *vaut* 1 => **y** *vaut* 0+1=1
**x** *vaut* 2 => **y** *vaut* 1+2=3
**x** *vaut* 3 => **y** *vaut* 3+3=6
**x** *vaut* 4 => **arrêt**

*(à la fin : x=4, y=6)*

### *4.6 Instruction case…of*

C'est une instruction de choix




<***expression***> doit être de l'un des types : integer, char, boolean, énuméré, intervalle .
<***constante***> doit obligatoirement être du même type que <***expression***>
<***Instruction***> est un bloc d'instruction simple ou composée (**begin** ..... **end**).

**Sémantique:**

C'est une instruction structurée équivalente à une série de **if**...**then**...**else** imbriqués. Cette instruction lorsque cela est possible, doit être préférée à un emboîtement de **if**...**then**...**else** dont la lisibilité n'est en fait pas optimale.

| **if**...**then**...**else** imbriqués | **case ... of** équivalent |
|---|---|
| **if** x = 3 **then** E1 **else**<br>**if** x = 4 **then** E2 **else**<br>**if** x = 5 **then** E2 **else**<br>**if** x = 6 **then** E2 **else**<br>**if** x = -5 **then** E3 **else** Ef | **case** x **of**<br>3 : E1 ;<br>4..6 : E2 ;<br>-5 : E3 ;<br>**else** Ef<br>**end** |

*Exemple :*

```
program CaseOf ;
var   x, y  : integer ;
   begin
     y :=1 ;
     for x := 0 to 4 do
        case x+1 of
          0..3 : y :=y*2 ;
          4 : y := y+100
          else y:=0 ;
        end
   end.
```

**y** *vaut* 1

*Exécution du case dans la boucle :*

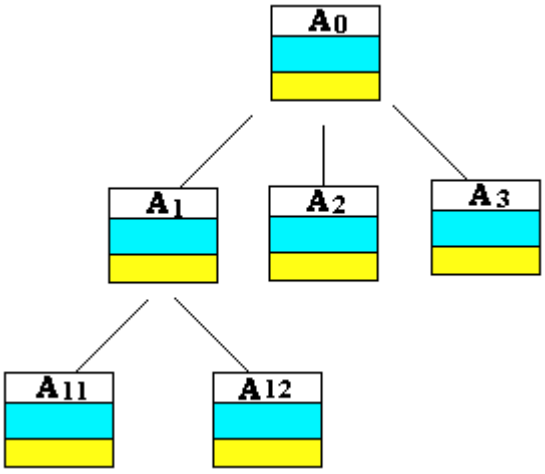
**x** *vaut* 0 => **x+1** *vaut* 1 (dans 0..3) => **y** *vaut* 1*2=2
**x** *vaut* 1 => **x+1** *vaut* 2 (dans 0..3) => **y** *vaut* 2*2=4
**x** *vaut* 2 => **x+1** *vaut* 3 (dans 0..3) => **y** *vaut* 4*2=8
**x** *vaut* 3 => **x+1** *vaut* 4 => **y** *vaut* 8+100=108
**x** *vaut* 4 => **x+1** *vaut* 5 (else) => **y** *vaut* 0
**x** *vaut* 5 => **arrêt**

*( à la fin : x=4, y=0 )*

# 5. Fonctions et procédures en pascal

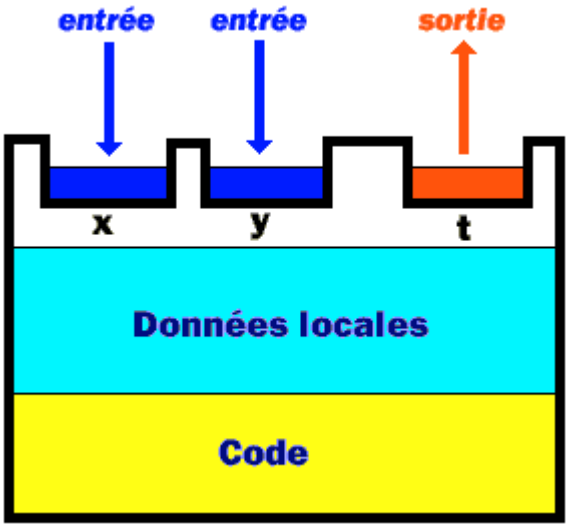
Le langage Pascal a été conçu à l'origine comme un langage pédagogique d'implantation de la programmation de type algorithmique; grâce à son extension objet Delphi il est utilisé comme outil de développement professionnel en entreprise.

La programmation algorithmique est une programmation hiérarchisée descendante.

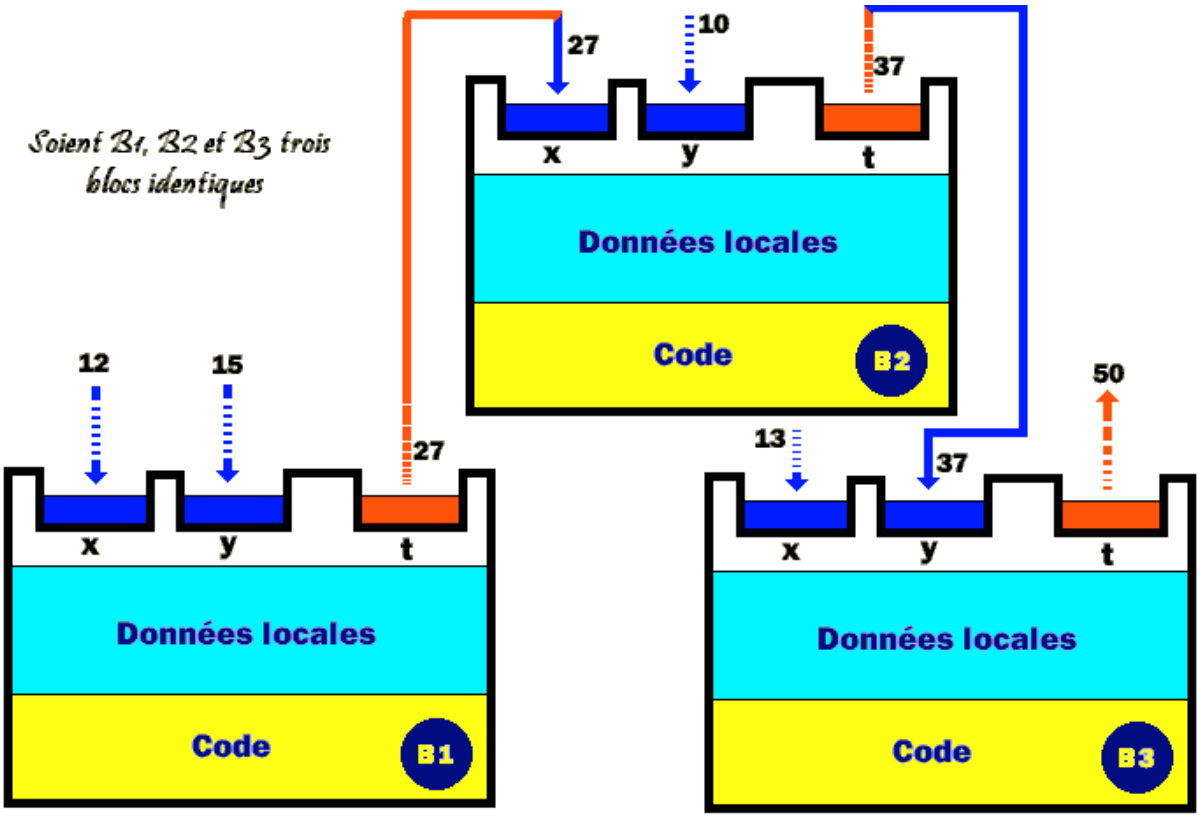



Cette décomposition descendante hiérarchique est construite à l'aide de blocs de programme notés aussi des sous-programmes.

Un bloc comporte donc des données locales, du code (instructions ou corps du bloc), des données d'entrée et/ou des données de sortie (permettant les échanges d'informations entre les différents blocs de la hiérarchie) :



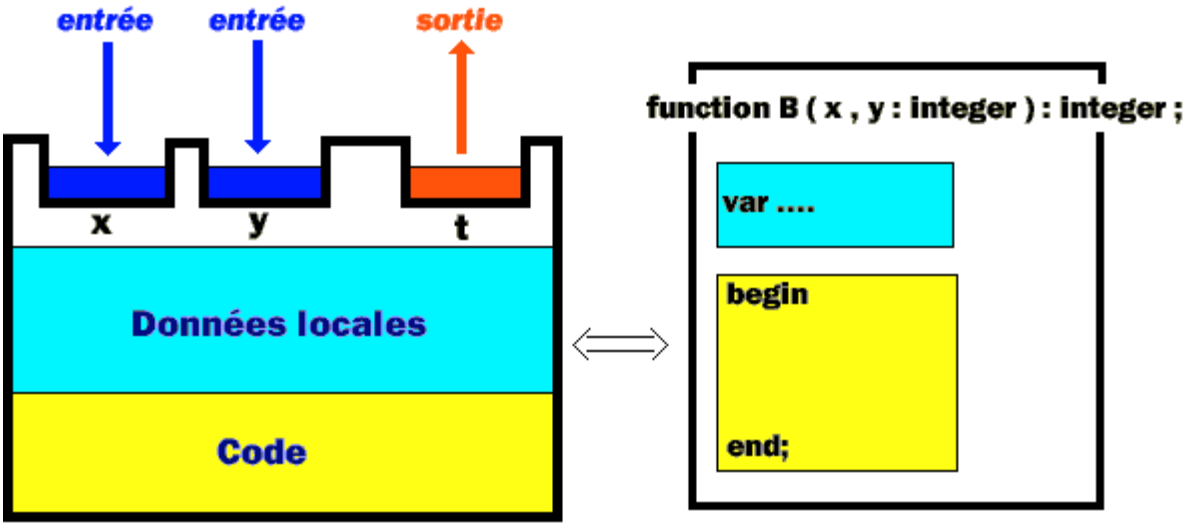

L'exemple ci-après représente trois blocs B1, B2 et B3 échangeant des informations (en fait chacun calcule la somme des deux entiers qu'il reçoit en entrée et renvoie leur somme :



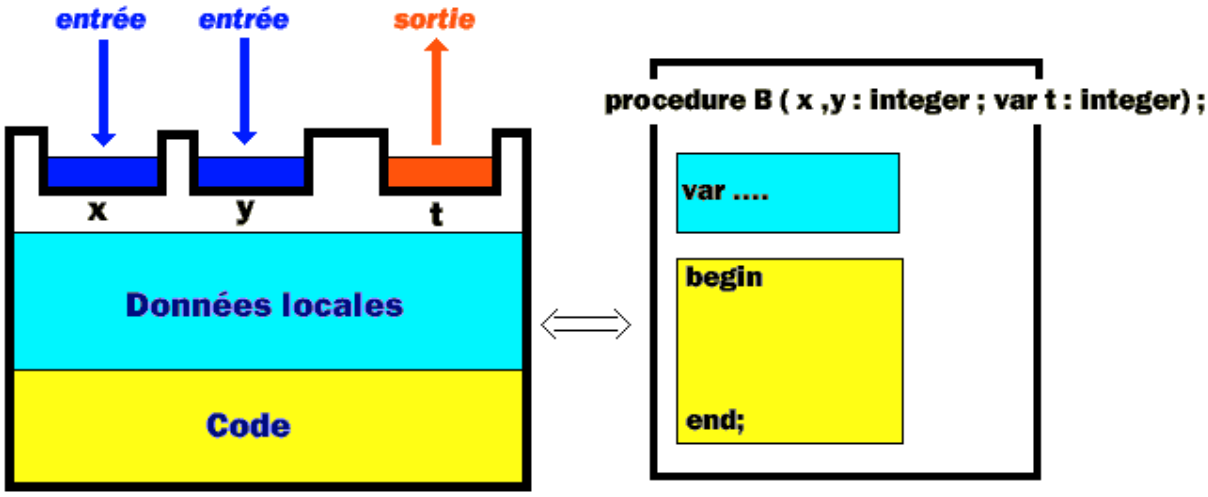

Le bloc B1 reçoit en entrée 12 et 15 et renvoie la somme 12+15 = 27 vers le bloc B2, la valeur 27 devient une donnée d'entrée pour le bloc B2 qui reçoit comme autre entrée la valeur 10. Le bloc B2 renvoie vers le bloc B3 le résultat 27+10 = 37 etc...

Nous remarquons que chaque bloc est indépendant des autres blocs. La seule liaison qui intervienne ici se situe dans le passage des données d'un bloc vers un autre bloc. Le code et les données locales d'un bloc fixé sont inaccessibles aux autres blocs.

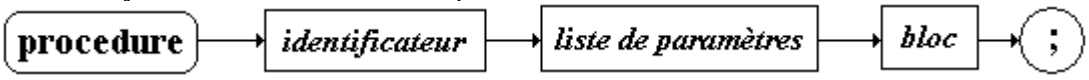
En pascal les blocs sont implémentés soit par des fonctions :



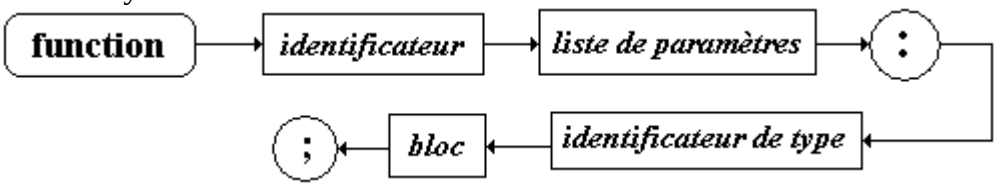
En pascal les blocs sont implémentés aussi par des procédures :




Voici la syntaxe de déclaration des procédures en Pascal :




Voici la syntaxe de déclaration des fonctions en Pascal :




- <***identificateur***> est le nom de la procédure ou de la fonction (choisi par vous)
- <***liste de paramètres***> est soit vide, soit elles contient entre parenthèses et séparés par des point-virgules la liste des paramètres formels.
- <***bloc***> est une instruction composée (**begin** ..... **end**).
- <***identificateur de type***>, dans le cas d'une fonction représente le type du résultat renvoyé par la fonction.

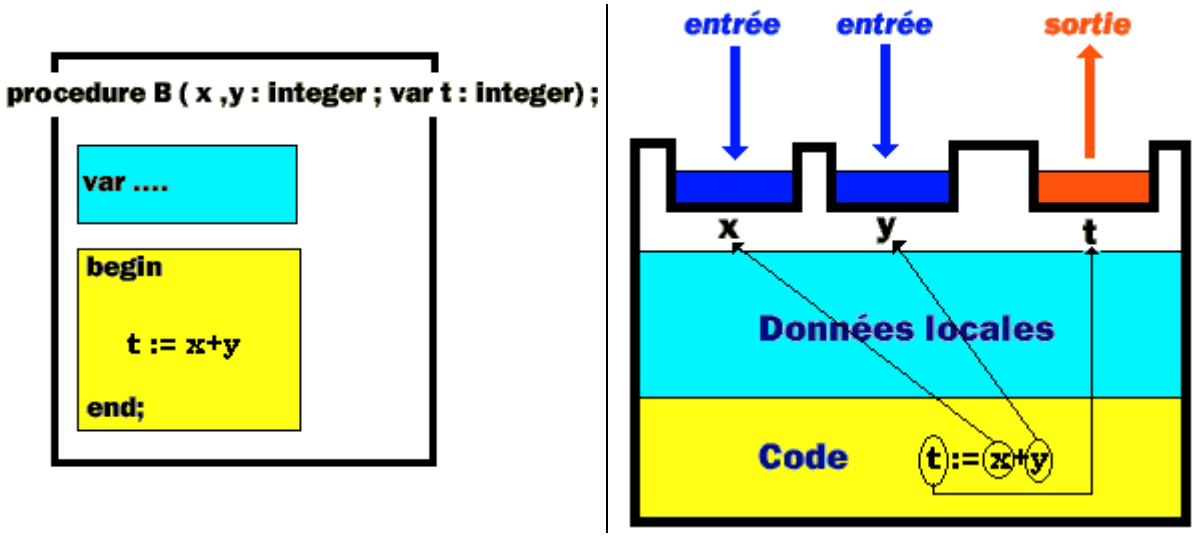
*Exemples de déclarations avec et sans paramètres formels :*

| | |
|---|---|
| **procedure** Somme (x,y :integer; **var** z :integer) ;<br>**begin**<br>z := x +y<br>**end ;** | **function** Somme (x,y :integer): integer ;<br>**begin**<br>result := x +y<br>**end ;** |
| **procedure** Somme ;<br>**var** x, y : integer ;<br>**begin**<br>y :=1 ; x := 2;<br>**writeln( x+y)**<br>**end ;** | **function** Somme : integer ;<br>**var** x, y : integer ;<br>**begin**<br>y :=1 ; x := 2;<br>result := x +y<br>**end ;** |

# 6. Paramètres en pascal

On s'intéresse dans ce paragraphe aux rapports qu'il y a entre un programme appelant et un sous-programme appelé uniquement en Pascal.

Soit par exemple une procédure B ayant 3 paramètres formels et renvoyant dans le troisième paramètre la somme des deux premiers :




Les paramètres formels d'une procédure jouent le rôle de variables muettes et servent à décrire le fonctionnement d'une procédure. Ils ont la même utilisation qu'une variable dans un polynôme mathématique. Les deux écritures $P(x) = 3x^2 - 4x + 5$ et $P(t) = 3t^2 - 4t + 5$ représentent mathématiquement le même polynôme, il en est de même pour une procédure.

on peut changer tous les paramètres formels d'une procédure sans en changer son fonctionnement

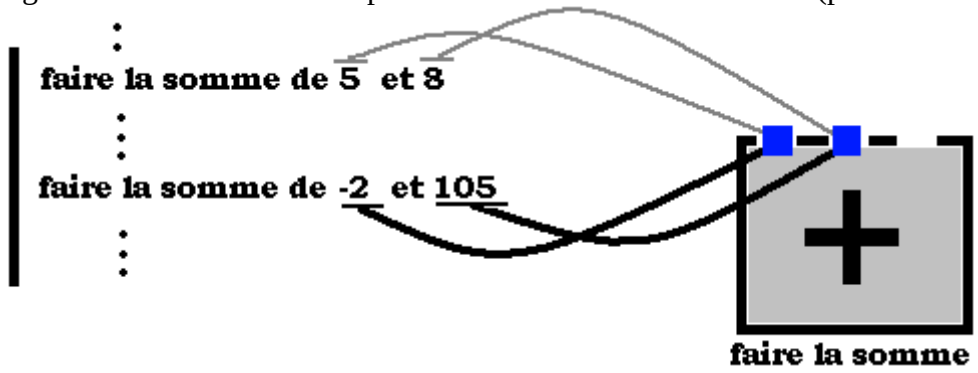
Les deux déclarations ci-dessous sont identiques :

```
procedure B (x , y :integer; var t :integer) ;
begin
  t := x +y
end ;

procedure B (a , b integer; var c :integer) ;
begin
  c := a +b
end ;
```

L'intérêt pratique d'une procédure et en général d'un sous-programme est essentiellement de pouvoir exécuter toujours la même action mais avec des valeurs différentes.

Par exemple une procédure P qui utilise un autre procédure B qui fait la somme, de deux entiers. La procédure B fonctionne comme une sorte de boîte noire qui reçoit deux valeurs en entrée et qui retourne leur somme comme dans le pseudo-code ci-dessous :

Lignes fictives de code de la procédure P utilisant la boîte noire (procédure B)



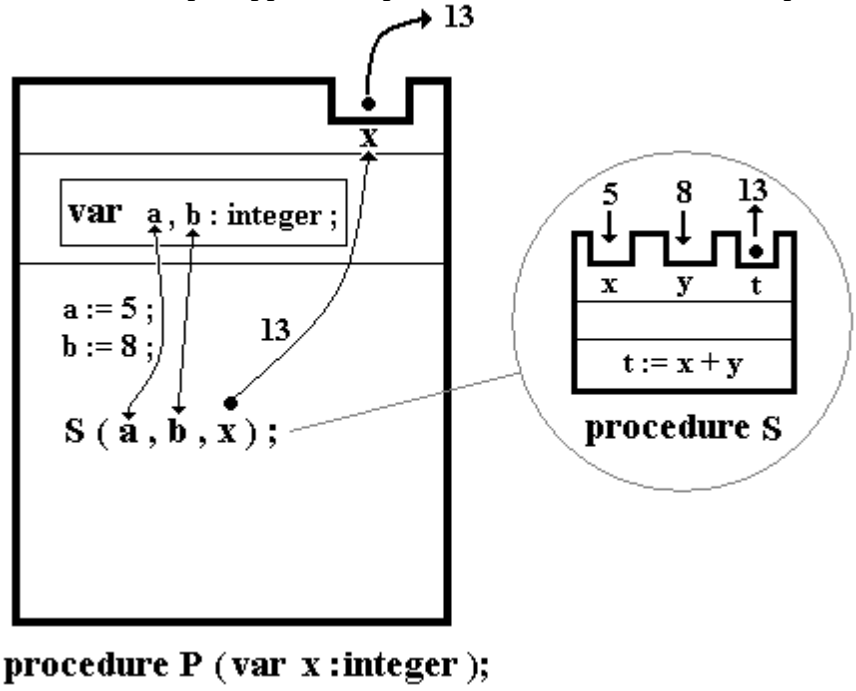

La boîte "faire la somme" est utilisée une première fois pour sommer 5 et 8, puis plus loin elle est utilisée une deuxième fois pour sommer -2 et 105.

Le mécanisme qui permet d'utiliser la procédure B dans le code de la procédure P se dénomme l'**appel** de procédure. P se dénomme la procédure **appelante**.

Reprenons l'exemple du polynôme écritures $P(x) = 3x^2 - 4x + 5$ , nous savons qu'en donnant une valeur effective à la variable x (par exemple x = 2) on obtient un résultat noté P(2) qui vaut: $P(x) = 3.2^2 - 4.2 + 5 = 9$.

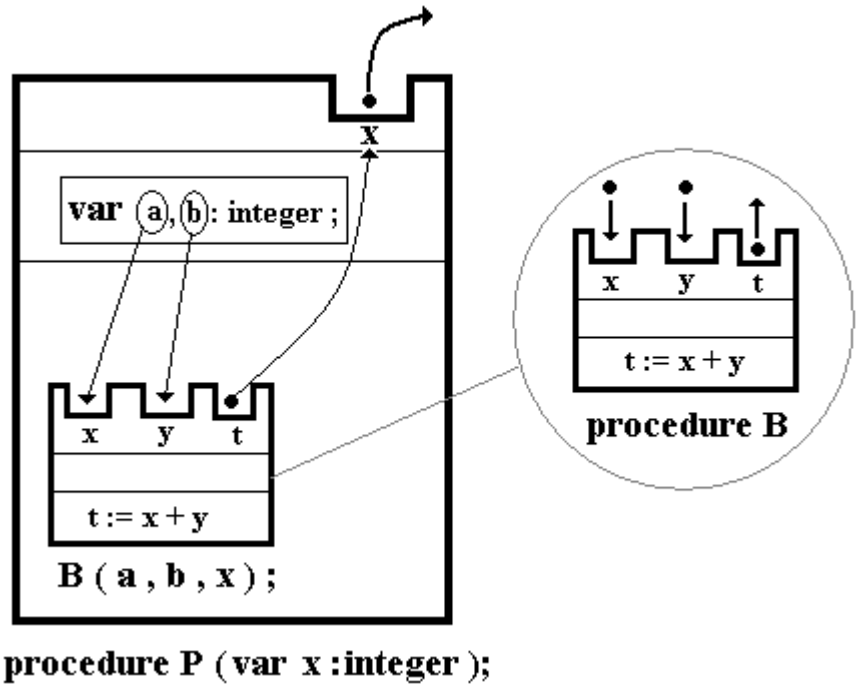
L'appel de procédure est un procédé très semblable au calcul du polynôme sur une valeur. La procédure a besoin qu'on lui fournisse des variables contenant effectivement des valeurs. De telles variables se dénomment les **paramètres effectifs** de la procédure.

Précisons un peu plus l'utilisation d'une procédure S avec des variables. Supposons que S serve à calculer la somme de deux valeurs **5** et **8** contenues respectivement dans deux variables locales **a** et **b** d'une autre procédure nommée P dont le seul paramètre **x** renvoie le résultat **13** du calcul obtenu par appel de la procédure B dans le code de la procédure P :



L'appel S(a,b,x) s'effectue sur des paramètres effectifs qui sont nécessairement des variables existantes, soit déclarées dans un paragraphe var dans la zone des données locales, soit déclarées en tant que paramètres de la procédure appelante.

L'appel se fait avec un nombre de paramètres effectifs égal à celui des paramètres formels en respectant l'ordre et la cohérence des types. On peut imaginer que lors d'un appel à la procédure S par le code de la procédure S, le code de la procédure S vient s'imbriquer *fictivement* dans le code de P à l'endroit de l'appel avec comme variables les paramètres effectifs :




## Comment a lieu cet appel, cette inclusion fictive du code ?

On dénomme l'action qui consiste à appeler sur des paramètres effectifs, le **passage** des paramètres effectifs ou encore la **transmission** des paramètres effectifs.

Il faut savoir qu'un **paramètre effectif** transmis au sous-programme appelé est un **moyen d'utiliser ou d'accéder** à une information appartenant au bloc appelant (le bloc appelé peut être le même que le bloc appelant, il s'agit alors de récursivité).

Pascal ne dispose que de 2 modes de passage sur les 5 modes généraux :

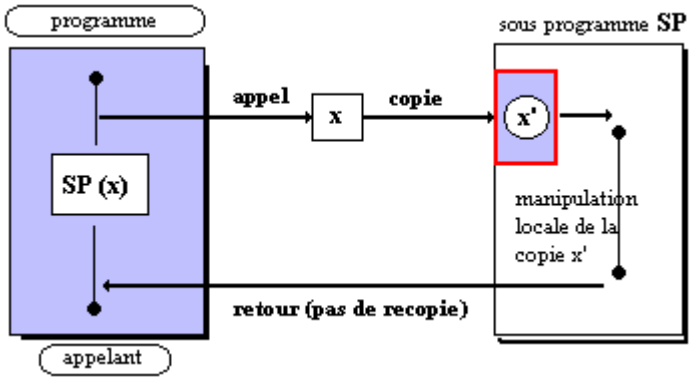
- Le passage par **valeur,**
- le passage par **référence** ou **adresse.**

### *6.1 Lecture seulement : passage par valeur*

Dans un passage par valeur, le paramètre formel est considéré comme une variable locale dans le corps du sous-programme. Sa valeur est initialisée au début de chaque exécution du sous-programme avec la valeur du paramètre effectif correspondant.

Il y a recopie de la valeur du paramètre effectif dans une zone spécifique locale à la procédure. Toutes les opérations qui sont effectuées sur le paramètre formel n'affectent que cette valeur locale.

Ecriture en Pascal **procedure** sp(... x: **real** ....)

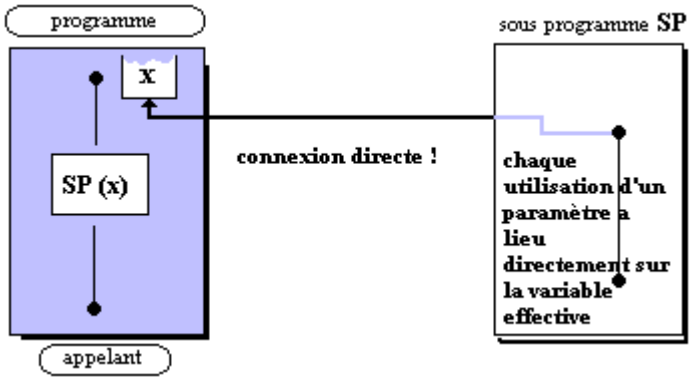



*passage par valeur*

### *6.2 Accès direct : passage par adresse ou par référence*

Dans un passage par adresse le paramètre formel est traité comme une variable dont l'adresse qui est transmise au moment de chaque appel, est celle du **paramètre effectif** correspondant. L'adresse de la variable effective autorise toutes les modifications immédiatement sur cette variable quelle que soit sa localisation.

Ecriture en Pascal **procedure** sp (....**var** x : **real** .....)




*passage par référence*

## Comparaison des avantages et des inconvénients des 2 modes

*Passage par valeur :*

- **Avantage** : sécurité et protection des informations.
- **Inconvénient** : lenteur due à la recopie des données et doublement de la place mémoire occupée (mais convient bien pour des variables simples !).

*Passage par référence :*

- **Avantage** : rapidité d'accès aux données, moindre occupation mémoire puisqu'il ne s'agit que d'une adresse.
- **Inconvénient** : ce mode est dangereux à cause de la non protection des données et de la nécessité qu'il y a de connaître la façon dont sont implantées physiquement les données sur la machine.

Ces deux modes de passage des paramètres sont présents dans des langages comme C++, java, Ada, Visual-Basic .net, Delphi et C#. Il suffit donc pour le débutant, de bien comprendre le processus avec le pascal et par analogie il pourra l'utiliser avec les autres langages.

*Exemple* :

```
procedure B1 (x : integer; var y : integer) ;
begin
  y := 10*x
end ;

procedure B2 (x : integer; y :integer) ;
begin
 y := 10*x
end ;

procedure P ;
  var a , b: integer ;
begin
  a := 100 ; b := 0 ;
  B1 ( a , b ) ;

  a := 100 ; b := 0 ;
  B2 ( a , b ) ;

end ;
```

*Dans la procédure B1*
*x est passé par valeur*
*y est passé par référence*

*Dans la procédure B2*
*x est passé par valeur*
*y est passé par valeur*

*Dans la procédure P*

*Appel de B1*
*B1( valeur a , ref b)*
*Résultat après appel :*
*b = 1000*

*Appel de B2*
*B2( valeur a , valeur b)*
*Résultat après appel :*
*b = 0*

## 7. Fonction ou procédure ?

Une fonction est un bloc de programme qui réalise des traitements et renvoie une valeur unique, c'est une procédure ne possédant qu'un seul élément de sortie (appelé paramètre).

**Tout** ce qui a été énoncé sur les procédures **s'applique** in extenso aux **fonctions**.

La ligne : "**procedure B ( x , y : integer ; var t : integer) ;** "
se dénomme l'en-tête de la procédure.

La ligne : "**function B ( x , y : integer ) : integer ;** "
se dénomme l'en-tête de la fonction

En pascal les blocs peuvent être implémentés aussi par des fonctions mais **uniquement** lorsqu'il n'y a qu'une donnée de sortie (un seul résultat).

*Exemple1* :

```
function B1 (x : integer ) : integer ;
begin
  result := 10*x
end ;

procedure P ;
  var
     a , b: integer ;
begin
  a := 100 ;
  b := B1 ( a )
end ;
```

*Dans la function B1*
*x est passé par valeur*
*B1 renvoie un résultat de type integer*

*Dans la procédure P*

*Appel de la fonction B1*
*b := B1( valeur a )*
*Résultat après appel :*
*b = 1000*

*Exemple2* :

```
function TTC (PHT,Tva : real ) : real ;
begin
  result := PHT*Tva
end ;


procedure CalculPrix ;
  var  PrixHT , PrixTTC : real ;
begin
 PrixHT:= 100 ;
 PrixTTC := TTC ( PrixHT , 1.186 )
end ;
```

*Dans la function TTC*
*PHT et Tva sont passés par valeur*
*TTC renvoie un résultat de type real qui est*
*Le paramètre prix hors taxe multiplié*
*par le paramètre taux de TVA.*

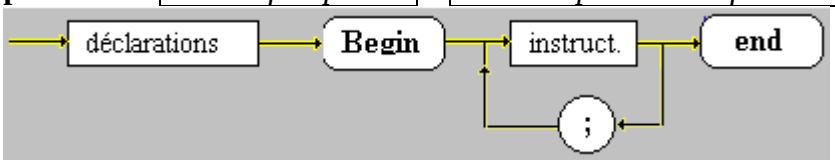
*Dans la procédure* CalculPrix

*PrixHT = 100 €*
*Appel de la fonction TTC*
*PrixTTC := TTC ( valeur a , 1.186 )*
*Résultat après appel :*
*PrixTTC = 118,6 €*

Les déclarations de fonctions et de procédures suivent le schéma grammatical de la déclaration générale du programme principal :

## *< Déclaration de procédure >*

**procedure** *<identif de proc>* *<liste de paramètres formels>* **;**



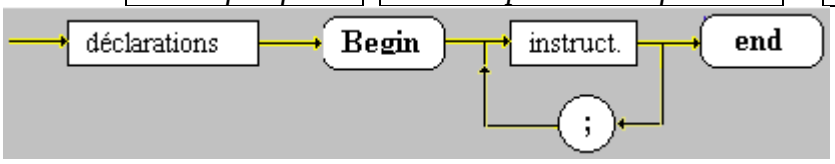

**;**

*Exemple :*

```
procedure    Calcul      (x : integer; var y :integer)   ;
             <identif de proc>   <liste de paramètres formels>

  var a, b : integer ; <déclarations>

  begin

    x := a*x  ; <instruction>
    y := x - b  ; <instruction>

  end  ;
```

## *< Déclaration de fonction >*

**function** *<identif de fonc>* *<liste de paramètres formels>* **:** *<type du résultat>* **;**




**;**

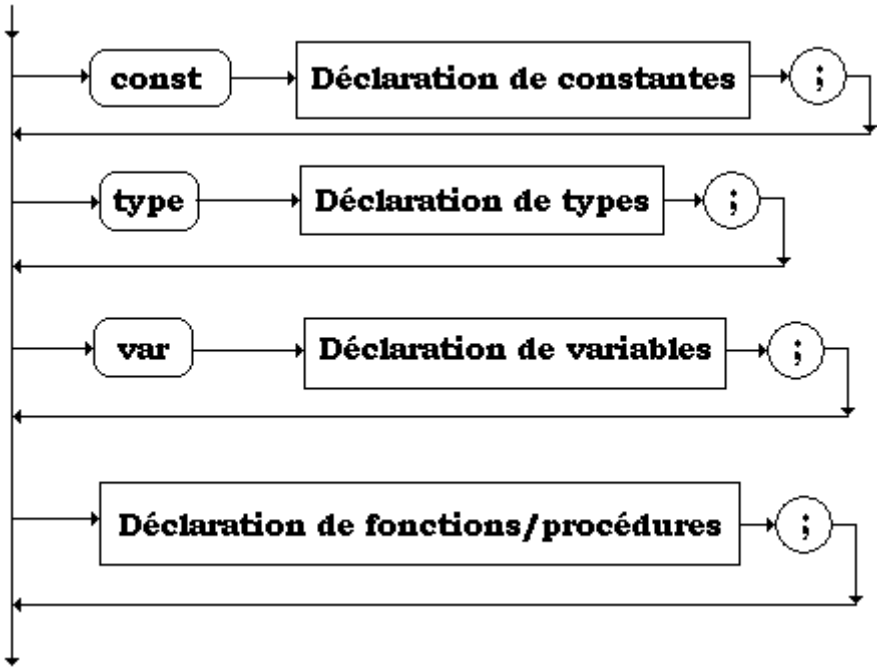
*Exemple :*

```
function     Calcul      (x : integer)     :     integer  ;
             <identif de fonc>  <liste de paramètres formels>   <type du résultat>

  var a : integer ; <déclarations>

  begin

    result := a*x  ; <instruction>

  end  ;
```

## 8. Visibilité des variables

Le langage pascal suivant méthode de la programmation structurée descendante, les déclarations de fonctions/procédures peuvent être imbriquées :

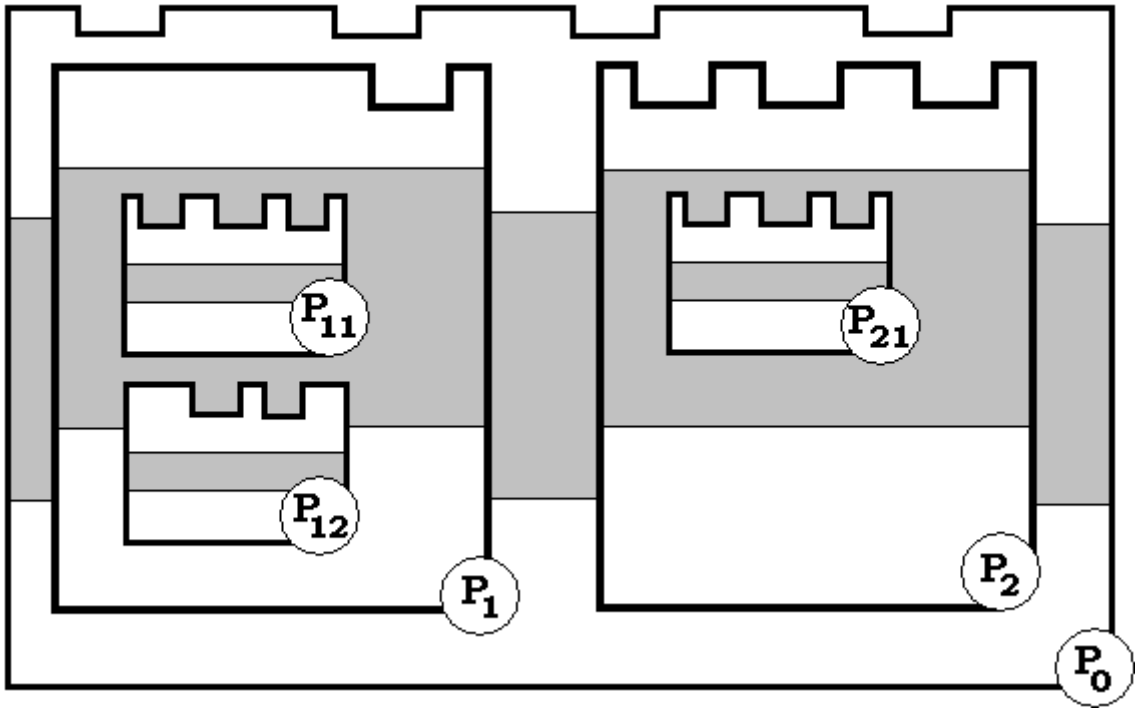
### *< Déclarations > :*



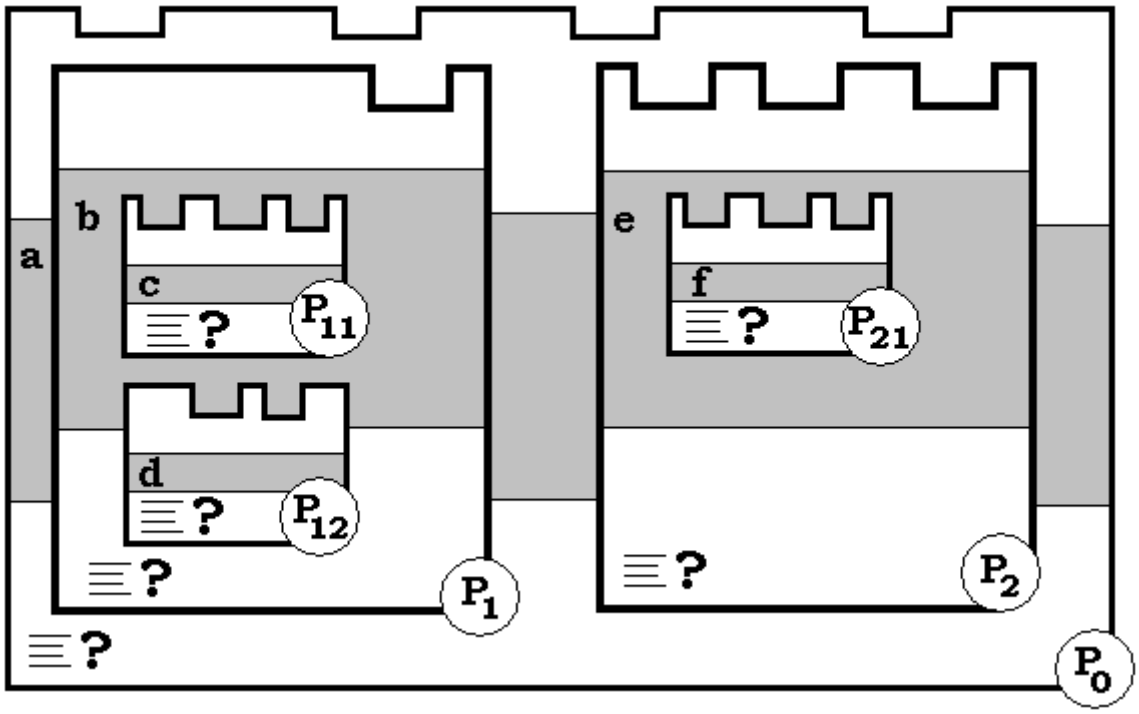

*Exemple de déclarations imbriquées dans la même procédure P0 :*

```
procedure P0 (x,y,z : char) ;
 var  a , b: integer ;

   procedure P1 ( var u : integer) ;
    var  a , b: integer ;
      procedure P11 ( var u,v,w : integer) ;
       var  a , b: integer ;
      begin
        …..
      end ;

      procedure P12 ( t : integer; h :char) ;
       var  a , b: integer ;
      begin
        …..
      end ;
   begin { P1 }
      …..
   end ; { P1 }

  procedure P2 ( f, g, h : real) ;
    var  a , b: integer ;
      procedure P21 ( n, m, p : integer) ;
        var  a , b: integer ;
      begin
        …..
      end ;

    begin  { P2 }
      …..
    end ; { P2 }

begin  { P0 }
  …..
end ; { P0 }
```

Ecritures que l'on peut représenter schématiquement par les imbrications de blocs qui suivent (les parties grisées d'un bloc correspondent à la partie déclaration du bloc) :




Supposons dans l'exemple précédent que la partie déclaration de chaque bloc contienne outre l'éventuelle déclaration d'un autre bloc, des déclarations de variables (**a** dans le bloc P0, **b** dans le bloc P1, **c** dans le bloc P11, **d** dans le bloc P12, **e** dans le bloc P2, **f** dans le bloc P21 ) :



Code pascal du schéma précédent :

```
procedure P0 (x,y,z : char) ;
 var  a : integer ;

   procedure P1 ( var s : integer) ;
    var  b : integer ;
      procedure P11 ( var u,v,w : integer) ;
       var  c : integer ;
      begin
        ….. ?….
      end ;

      procedure P12 ( t : integer; h :char) ;
       var  d : integer ;
      begin
        ….. ?….
      end ;
   begin  { P1 }
      ….. ?….
   end ; { P1 }

  procedure P2 ( f, g, h : real) ;
    var  e : integer ;
      procedure P21 ( n, m, p : integer) ;
        var  f : integer ;
      begin
        ….. ?….
      end ;

    begin  { P2 }
      ….. ?….
    end ; { P2 }

begin  { P0 }
  ….. ?….
end ; { P0 }
```

Etant donné les possibilités offertes par cette disposition des blocs en Pascal, il vient immédiatement une question sur les accès autorisés ou non aux données situées dans les parties déclarations des blocs P0, P1, etc...

En d'autres termes, dans la partie code de chaque bloc quelles variables peut-on utiliser ? Par exemple dans le corps (la partie code) de la procédure P12 peut-on utiliser toutes les variables **a, b, c, d, e, f** ou bien seulement certaines et selon quelles règles ?

```
procedure P12 ( t : integer; h :char) ;
  var  d : integer ;
 begin
   ….. ?….
 end ;
```

Ces autorisations d'accès aux données situées dans des blocs imbriquées sont contenues dans la notion de règle de visibilité dans les langages à structure de bloc (Pascal en est un cas particulier, ces règles s'appliqueront aussi à d'autres langages)

## Règle de visibilité:

Toute donnée X déclarée localement dans un bloc $P_k$ est n'est visible que :

- dans le bloc où elle est déclarée,
- et dans tous les blocs $P_{k+n}$ imbriqués dans $P_k$.
- Un paramètre formel est considéré comme une variable locale au bloc.

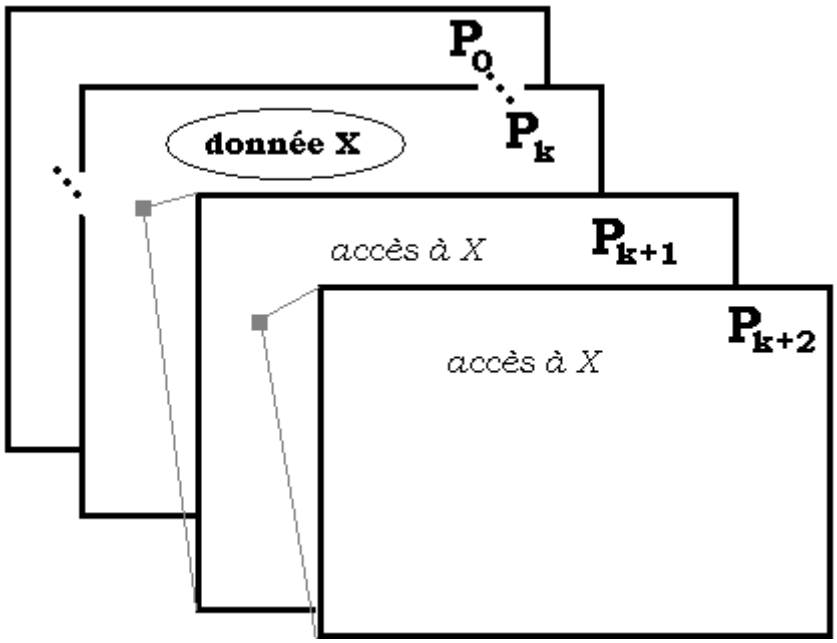



*fig - visibilité d'une donnée X déclarée dans $P_k$*

**Remarque : masquage**

Lorsqu'une donnée déclarée sous le nom X dans un bloc $P_k$ est redéclarée sous le même nom X dans un bloc $P_{k+n}$ imbriqué dans $P_k$, la donnée X de $P_{k+n}$ masque les informations contenues dans la donnée X de $P_k$ dans le bloc $P_{k+n}$ et dans ceux qu'ils contient.

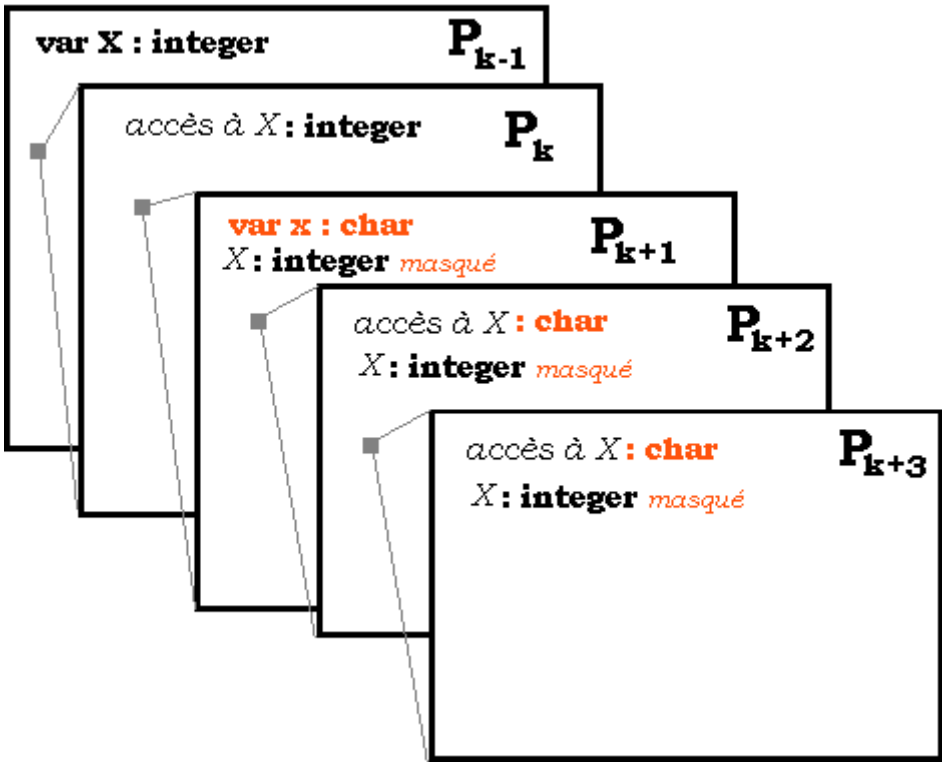



*fig - visibilité d'une donnée X déclarée dans $P_{k-1}$*

Etudions la visibilité des variables **a, b, c, d, e, f** dans les blocs P0, P1, P11, P12, P2, P21 figurées ci-dessous :

```
procedure P0 (……) ;
 var  a : integer ;

   procedure P1 ( …….) ;
    var  b : integer ;
      procedure P11 ( ……) ;
       var  c : integer ;
      begin
        ….. ?….
      end ;

      procedure P12 ( ……) ;
       var  d : integer ;
      begin
        ….. ?….
      end ;
   begin  { P1 }
      ….. ?….
   end ; { P1 }
```

```
  procedure P2 ( ……. ) ;
    var  e : integer ;
      procedure P21 ( …….. ) ;
        var  f : integer ;
      begin
        ….. ?….
      end ;

    begin  { P2 }
      ….. ?….
    end ; { P2 }

begin  { P0 }
  ….. ?….
end ; { P0 }
```

Etablissons à partir de la règle de visibilité énoncée plus haut, deux tableaux récapitulatifs croisés de la visibilité des variables **a, b, c, d, e, f** :

| variable | Bloc où cette variable est visible |
|---|---|
| **a** | P0, P1, P11, P12, P2, P21 |
| **b** | P1, P11, P12 |
| **c** | P11 |
| **d** | P12 |
| **e** | P2, P21 |
| **f** | P21 |

| Bloc | variables visibles dans ce bloc |
|---|---|
| P0 | **a** |
| P1 | **a, b** |
| P11 | **a, b, c** |
| P12 | **a, b, d** |
| P2 | **a, e** |
| P21 | **a, b, f** |

Nous pouvons donc répondre maintenant aisément à la question posée plus haut : quelles variables peut utiliser dans la procédure P12 ?

La procédure P12 accède aux variables **a, b** et **d**, ( avec en plus comme variables locales ses paramètres formels **t** et **h** ) :

```
procedure P12 ( t : integer; h :char) ;
  var  d : integer ;
 begin
   // accès aux variables a, b, d, t et h,
 end;
```

# 9. Variables dynamiques, références ou pointeurs

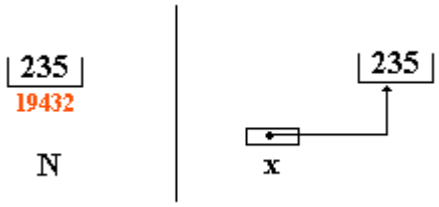
## Définition

Beaucoup de langages disposent de la notion de pointeur C++ en particulier. C'est une notion proche de la machine qui a été utilisée dès le début pour représenter dans un programme l'allocation dynamique de mémoire. Dans une structure à allocation dynamique de mémoire le compilateur ne connaît pas à l'avance la taille de la structure, la gestion de la mémoire est alors confiée au programmeur. C'est lors de l'exécution et au fur et à mesure des mises à jours que la taille de la structure varie, comme par exemple dans la gestion d'une liste dont la taille varie en fonction des ajouts ou des suppressions. A l'opposé, une structure statique est une entité dont le compilateur connaît très exactement la taille avant l'exécution du programme, comme par exemple la structure de données de type tableau peut être considérée comme une structure statique puisque la taille du tableau (nombre de cellules) est connue lors de la déclaration.

En fait, les langages récents ne disposent plus de cette notion de pointeurs ou variables dynamiques parce qu'à l'usage elle s'est révélée dangereuse car trop proche de la machine laissant le programmeur se débrouiller seul avec la gestion de la mémoire, elle est utilement remplacée par la notion de référence d'objet comme dans Java, le langage C# demandant une autorisation pour traiter du code non sûr (**unsafe** code). Delphi quant à lui, combine les deux outils : pointeurs et références d'objet,la version Delphi 8.Net adoptant la même démarche que C# (**unsafe** code).

La notion de pointeur très présente, voir même essentielle dans un langage comme le C, est utilisable en pascal.

Prenons par exemple une variable numérique N entière d'adresse en mémoire centrale 19432 et contenant le nombre entier 235, nous appelons x un pointeur vers cette variable N, une variable dynamique contenant l'adresse de la variable N :



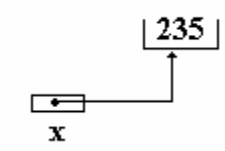

Nous dirons aussi que x « pointe » vers la variable N et que le « contenu » de x est 235.

En pascal (utiliser Delphi en mode console), une variable dynamique se déclare comme une variable classique mais le type est précédé du symbole « ^ », elle est typée (le type de la donnée vers laquelle elle pointe), mais sa gestion est entièrement à la charge du programmeur à travers les procédures d'allocation et de désallocation mémoire respectivement appelées **new** et **dispose**.

## Utilisation pratique des variables dynamiques

**Contenu d'une variable dynamique « x » déjà allouée :** il est noté **« x^ »**

Dans l'exemple précédent :

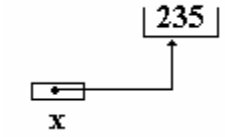



**x^ vaut 235** (contenu de la variable dynamique)

**x vaut 19432** (adresse de la variable dynamique)

## Détaillons pas à pas un programme d'utilisation de pointeur

Soit l'exemple précédent :




Le programme de droite écrit sur l'écran le « contenu » de la variable x (contenu de la cellule pointée par x) soit : **x vaut : 235**.

Voici le programme à analyser :

```
program VarDyn;
 var
  x : ^integer;
 begin
  new(x);
  x^:= 235;
  writeln('x vaut: ',x^);
  dispose(x);
 end.
```

**Déclaration d'une variable dynamique « x » de type entier :**

Soit l'instruction :

var x : ^integer ;

**Résultat produit :**

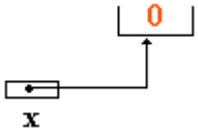

**x** est créée (mais x ne pointe vers rien encore)
**x vaut nil**

**Allocation d'une variable dynamique « x » déjà déclarée :**

Soit l'instruction :

new ( x ) ;

**Résultat produit :**



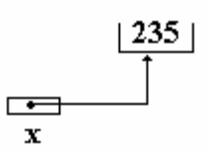

**une cellule mémoire de type integer est crée,**
**x** pointe vers la cellule créée.
(x vaut la valeur de l'adresse de la cellule)

**Affectation du contenu d'une variable dynamique « x » déjà déclarée :**

Soit l'instruction :

```
x^  :=  235 ;
```

**Résultat produit :**



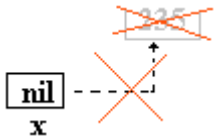

**La cellule mémoire pointée par x contient 235.**

**Désallocation d'une variable dynamique « x » déjà allouée :**

Soit l'instruction :

```
dispose ( x ) ;
```

**Résultat produit :**



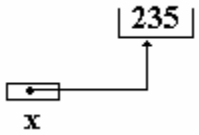

La cellule mémoire qui contenait 235 n'existe plus, elle est rendue au système (ont dit désallouée)

Attention

> Ne pas confondre l'effacement de l'adresse d'une variable dynamique et sa désallocation.

**Effacement de l'adresse d'une variable dynamique :** mot clef **« nil »**
**Désallocation d'une variable dynamique :** procédure **dispose(…)**

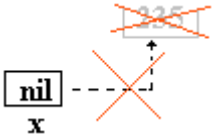
Soit l'exemple précédent :




**Résultat produit par** x := nil **:**


- **x^** n'existe plus (x ne pointe vers plus rien)
- **x vaut nil**
- La cellule mémoire qui contient 235 **existe toujours**, mais n'est plus accessible !

**Résultat produit par** dispose ( x ) **:**




- **x^** n'existe plus (x ne pointe vers plus rien)
- **x vaut nil**
- La cellule mémoire qui contenait 235 **n'existe plus** !

C'est en particulier cette dernière remarque qui pose le plus de soucis de maintenance aux développeurs utilisant les pointeurs (par ex : problème de la référence folle).

**Affectation de variables dynamiques entre elles :**

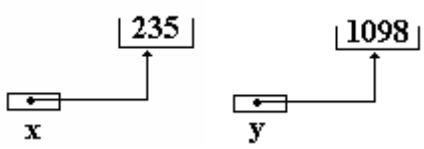
On suppose que deux variables dynamiques « x et y » de type ^**integer** ont été déclarées et créées par la procédure **new**, nous figurons ci-après l'incidence de l'affectation x := y sur ces variables :

Soient les instructions :

x^ := 235 ;

y^ := 1098 ;

**Résultat produit :**

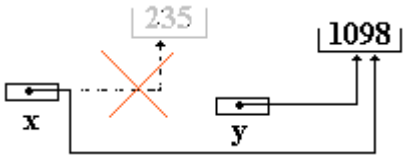



Soient l'affectation :

x := y ;

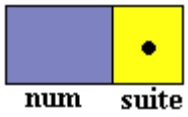
***x et y pointent vers la même cellule mémoire***

**Résultat produit :**

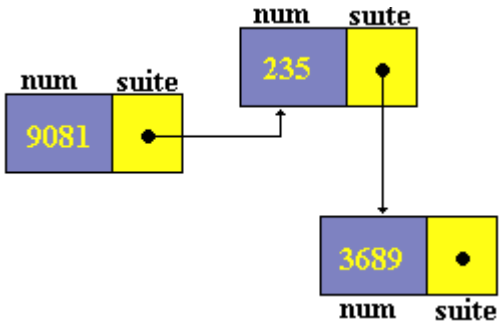



## Une structure de données récursive avec pointeurs

Prenons une structure de données organisée sous forme de liste composée de cellules qui sont elles mêmes chacune un enregistrement (un **record**) contenant deux champs **num** et **suite** :




Le champ **num** est de type entier, et le champ **suite** est une variable dynamique de type cellule (lorsqu'il est alloué, il pointe donc vers une nouvelle cellule) :

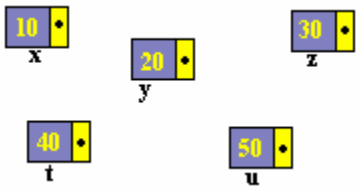


Soit un programme d'exemple de structure récursive (Delphi en mode console) utilisant les variables dynamiques pour représenter cette structure.

```
program pointeur;

 type

  cell = ^struct;

  struct = record
   num : integer;
   suite : cell
  end;

  var
   x , y, z , t , u : cell;
```

```
begin
   new(x) ;    x^.num := 10;
   new(y) ;    y^.num := 20;
   new(z) ;    z^.num := 30;
   new(t) ;    t^.num := 40;
   new(u) ;    u^.num := 50;
end.
```

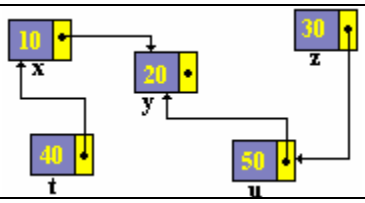
**Ce programme crée 5 cellules :**

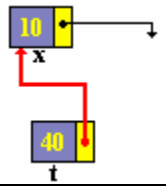
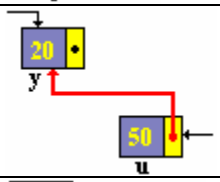
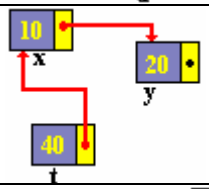
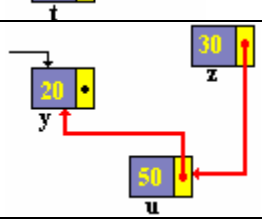



Les instructions suivantes :

```
t^.suite := x;
x^.suite := y;
z^.suite := u;
u^.suite := y;
```

représentent les liens ci-contre:




| L'instruction suivante | Représente l'accès au lien | Et écrit sur la console |
|---|---|---|
| writeln ( t ^. suite^. num); |  | 10<br>(le contenu du champ num de x) |
| writeln ( u^. suite^. num); |  | 20<br>(le contenu du champ num de y) |
| writeln ( t ^.suite^.suite^.num); |  | 20<br>(le contenu du champ num de y) |
| writeln ( z ^.suite^.suite^.num); |  | 20<br>(le contenu du champ num de y) |

La notion de référence est abordée au chapitre sur la programmation objet, c'est en fait un pointeur entièrement encapsulé sur lequel il n'est possible de faire qu'une seule opération : l'affectation de référence.

# 10. Récursivité en programmation

## Définition

Une famille d'objet est dite récursive, si dans sa définition il est fait référence à la famille elle-même.

Pour un langage de programmation, nous dirons qu'il autorise la récursivité si un sous-programme peut s'appeler lui-même directement ou indirectement à travers un autre sous-programme.

Le langage de programmation Algol 60 a été le précurseur sur le sujet de la récursivité. D'une manière générale un langage de programmation récursif doit donc être capable dans son implémentation, de conserver les contextes successifs provenant de chaque appel récursif du sous-programme.

Pour les langages à structure de bloc le problème de la conservation des contextes successifs est résolu grâce à la **pile d'exécution dynamique** : les variables locales et les paramètres sont empilés à chaque appel récursif du sous-programme.

*Récursivité directe et indirecte en Pascal-Delphi :*

<table>
<tr><th>Récursivité directe</th><th colspan="2">Récursivité indirecte ou croisée</th></tr>
<tr><td rowspan="2">Procedure P ;<br>Begin<br>….. P ;<br>End;</td><td>Procedure A ;<br>Begin<br>….. C ;<br>End;</td><td>Procedure B ;<br>Begin<br>….. A ;<br>End;</td></tr>
<tr><td colspan="2">Procedure C ;<br>Begin<br>….. B ;<br>End;</td></tr>
</table>

Notons que dans le cas de la récursivité croisée, il existe un problème syntaxique de déclaration d'une procédure avant l'autre :

<table>
<tr><td>Procedure A ;<br>Begin<br>….. B ;<br>End;</td><td>Procedure B ;<br>Begin<br>….. A ;<br>End;</td></tr>
</table>

La directive **forward** sert à résoudre ce problème. Lors de la déclaration, cette directive sert à **déclarer syntaxiquement l'en-tête** d'une procédure qui sera **déclarée en totalité plus loin**. Cette directive permet d'utiliser la récursivité croisée en particulier :

<table>
<tr><td>Procedure B ; forward ;<br><br>Procedure A ;<br>Begin<br>….. B ;<br>End;</td><td>Procedure B ;<br>Begin<br>….. A ;<br>End;</td></tr>
</table>

# Exemples en Pascal-Delphi de base

Le traitement de problème relatifs à des suites récurrentes ou de définition récurrentes (du genre $U_n = f(U_{n-1})$) peut s'effectuer à l'aide de la récursivité.

1°) Définition récursive de la fonction puissance entière $x^n$ :

$x^n = x^{n-1} * x$ , $\forall n \in N^*$

$x^0 = 1$

Implantation en Pascal-Delphi :

```
function puissance ( n : integer; x : real) : real ;

begin
  if n = 0 then result := 1
  else   result := x*puissance (n-1,x)
end;
```

2°) Définition récursive de la fonction factorielle du nombre entier n :

$n ! = (n-1)! * n$ , $\forall n \in N^*$

$0 ! = 1$

Implantation en Pascal-Delphi :

```
function fact ( n : integer ) : integer;

begin
  if n = 0 then result := 1
  else   result := x* fact (n-1)
end;
```

3°) Définition récursive du pgcd de 2 entiers **a** et **b** par la méthode d'Euclide :

$\forall a, a \in N^*$, $\forall b, b \in N^*$

pgcd ( a et b ) = pgcd ( b et reste (a par b) )

Implantation en Pascal-Delphi :
( *l'opérateur **mod** du pascal permet de calculer le reste de la division de a par b , on note : "a **mod** b"* )

```
function pgcd1 ( a,b : integer ) : integer;

begin
  if b = 0 then result := a
  else   result := pgcd1 (b, a mod b)
end;
```

4°) Définition récursive du pgcd de 2 entiers **a** et **b** par la méthode Egyptienne :

$\forall a, a \in N^*$, $\forall b, b \in N^*$
pgcd ( a et b ) = pgcd ( b , a-b ),  si $a \geq b$
pgcd ( a et b ) = pgcd ( a , b-a ),  si $b > a$

Implantation en Pascal-Delphi :

```
function pgcd2 ( a , b : integer ) : integer;

begin
  if a = b then result := a
  else  begin
            if a < b then result := pgcd2( a , b-a )
            else  result := pgcd2( b , a-b )
        end
end;
```

## 5°) Programmation récursive de l'inversion d'une chaîne de caractères :

Soit à construire une fonction qui reçoit une chaîne de type string et qui renvoie cette chaîne inversée.

Implantation en Pascal-Delphi :

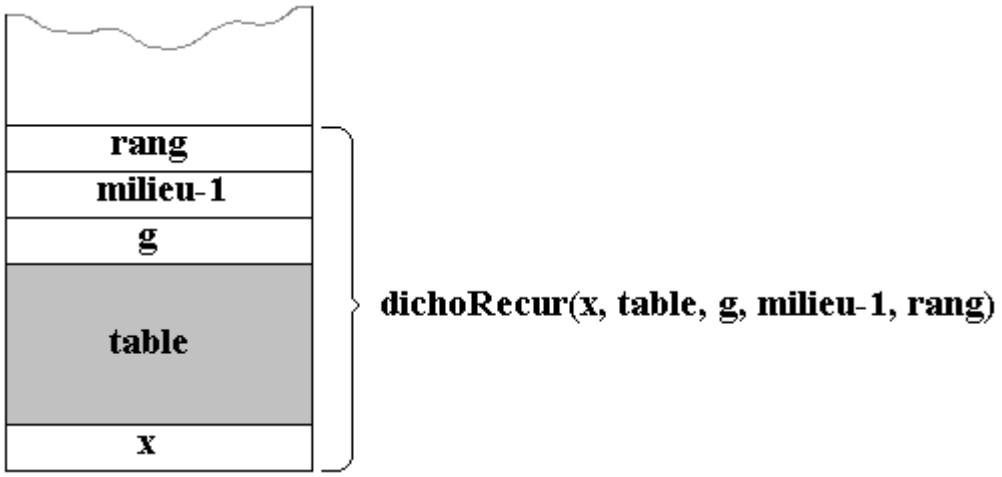
```
function InvCh ( ch : string ) : string;

begin
  if length(ch) < 2 then result := ch   // si ch est vide ou si ch n'a qu'un seul caractère
  else   result := InvCh ( Copy(ch , 2 , length(ch)-1 ) + ch[1]
end;
```

## 6°) Procédure récursive de recherche dichotomique dans un tableau trié :

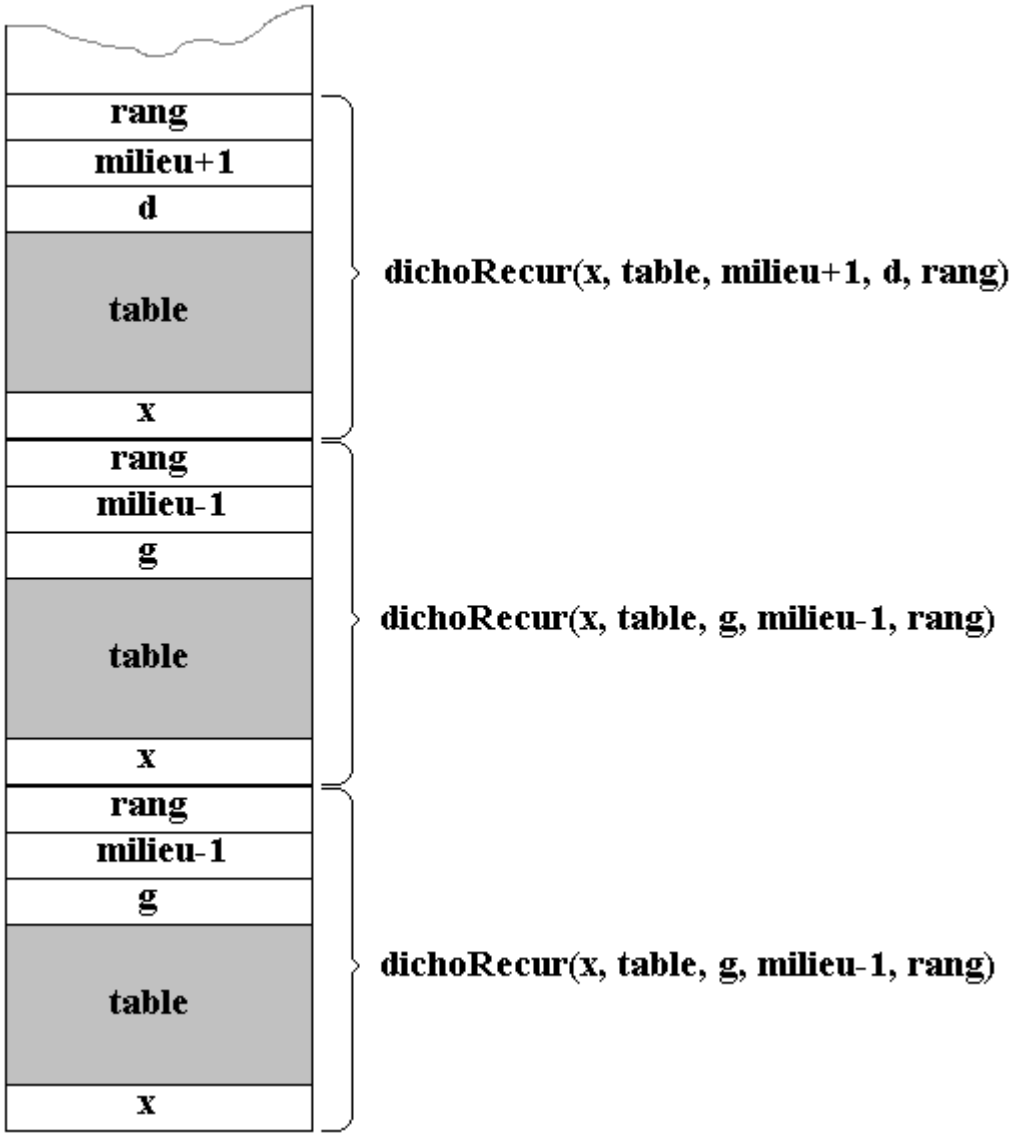
Soit à construire une procédure permettant de rechercher un élément **x** dans un tableau **table** et de renvoyer son rang ou -1 si l'élément n'est pas présent.

Implantation en Pascal-Delphi :

```
type
   Elmt = integer ;
   tableau = array[1..max] of Elmt ;

procedure dichoRecur(x : Elmt; table:tableau; g,d:integer; var rang:integer) ;
{ recherche dichotomique récursive dans table
  rang =-1 si pas trouvé.  g, d : 0..max+1}
var
  milieu:1..max;

begin
 if g <= d then
 begin
    milieu := (g+d) div 2;
    if x=table[milieu] then  rang:=milieu
    else
      if x < table[milieu] then dichoRecur(x, table, g, milieu-1, rang)
      else dichoRecur(x, table, milieu+1, d, rang)
 end
 else rang:=-1
end; {dichoRecur}
```




*Pile d'exécution du contexte du premier appel récursif de dichoRecur(x, table, g, milieu-1, rang)*




*Empilement des contextes de trois appels récursifs de la procédure dichoRecur :*
*dichoRecur(x, table, g, milieu-1, rang)*
*dichoRecur(x, table, g, milieu-1, rang)*
*dichoRecur(x, table, d, milieu+1, rang)*

# Exercices chapitre 2

*Ex-1 :* Au sujet des parenthèses bien formées dont on rappelle une C-grammaire :

G : $\mathbf{V_N} = \{S\}$
$\mathbf{V_T} = \{ (\ ,\ ) \}$
**Axiome** : S
**Règles** 1 : S ⟶ (SS)S
2 : S ⟶ ε

1°) Proposez 3 autres C-grammaires engendrant le même langage de parenthèses.
2°) Construisez dans chacune d'elle l'arbre de dérivation du mot **((( )( ))(( )).**

*Ex-2 :* Soit G la C-grammaire suivante et L(G) le langage engendré par G :

G : $V_N = \{ S, A, B \}$
$V_T = \{ \mathbf{(}\ ,\ \mathbf{)}\ ,\ \mathbf{o} \}$
**Axiome** : A
**Règles** :
1 : A → **(** A
2 : A → **(** S
3 : S → **o** S
4 : S → **)** B
5 : B → **)** B
6 : B → **)**

**1**°) Donnez le mot le plus petit appartenant au langage L(G) (celui de longueur minimale).
**2**°) Donnez très précisément la forme générale des mots du langage L(G) (avec contraintes sur les indices lorsqu'il y en a).
**3**°) construisez l'arbre de dérivation dans G de la chaîne : $\mathbf{(}^{3}\ \mathbf{o}^{2}\ \mathbf{)}^{4}$

*Ex-3 :* Problème classique du **défaut de fermeture** en Algol, en Pascal en Java, en C# et autre... :

A - Soit $G_0$ une grammaire ambiguë de l'instruction if ...then...else en Pascal
$\mathbf{V_N} = \{$<Expr.> , S$\}$
$\mathbf{V_T} = \{$ **if** , **then** , **else** , **P** , **a** $\}$
**Axiome** : S
**Règles** 1 : S ⟶ **if** < Expr.> **then** S
2 : S ⟶ **if** < Expr.> **then** S **else** S
3 : S ⟶ **a**
4 : < Expr.> ⟶ **P**

Donnez 2 arbres de dérivation dans $G_0$, de la chaîne :

**if** P **then if** P **then** a **else** a

B - On propose une autre grammaire ambiguë $G_1$ du même langage :
$\mathbf{V_N} = \{$<Expr.> , S , S'$\}$
$\mathbf{V_T} = \{$ **if** , **then** , **else** , **P** , **a** $\}$
**Axiome** : S
**Règles** 1 : S ⟶ **if** < Expr.> **then** S S'
2 : S ⟶ **a**
3 : S' ⟶ **else** S
4 : S' ⟶ ε

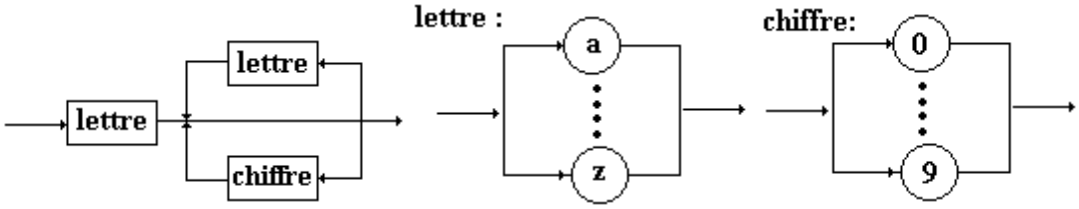
5 : < Expr.> ⟶ **P**

a) Refaire les 2 arbres de dérivation du mot « **if** P **then if** P **then** a **else** a » dans $G_1$.
b) Quelle amélioration est apportée par $G_1$ par rapport à $G_0$ ?

C - On construit une grammaire non ambiguë du langage :

a) Dans la grammaire $G_0$ , proposez une solution syntaxique permettant de lever l'ambiguïté en rajoutant un symbole supplémentaire dans $\mathbf{V_T}$ (grammaire $G_2$).
b) Montrer que le nouveau mot « **if** P **then if** P **then** .... » ne peut avoir qu'un seul arbre de dérivation dans $G_2$.

*Ex-4 :* ci dessous les diagrammes syntaxiques d'un identificateur dans un langage de programmation :




Ecrivez une grammaire en BNF traduisant ces diagrammes.

*Ex-5 :* Soit le programme Pascal suivant :

```
Program essai;
Const
  n = 50
Type
  tableau = array[1..n] of integer;
Var
  table : tableau ;

Procedure Lire( T : tableau );
begin
 for i:=1 to n do
  Readln( T[i] );
End;

begin
 Lire ( table);
 for i:=1 to n do
  write( T[i] ,'  ' );
end.
```

Que fait et qu'affiche très précisément ce programme ?

*Ex-6 :* Ecrire un programme Delphi console calculant la somme des 10 premiers termes de la série $S_n = \Sigma\ 1/(2i+1)$, soit :S=1+1/3+1/5+1/7+...+1/19.Le programme affichera la somme S.

*Ex-7 :* Delphi possède une fonction LowerCase permettant de transformer tous les caractères d'une string en minuscule. Ecrivez votre propre fonction "Lowerstring (nom:string)" qui renvoie la string nom en minuscule sans utiliser la fonction LowerCase.

*Ex-8 :* Delphi possède les opérateurs booléens or, and et Xor. Ecrire un programme console affichant la table de vérité de chacun de ces trois opérateurs.

*Ex-9 :* Ecrivez les fonctions booléennes : "implique(p , q: boolean)" qui renvoie le résultat de p => q et "equivalent(p , q: boolean)" qui renvoie le résultat de p ⇔ q.

**Réponses partielles:**

*Ex-1 :* **Règles**

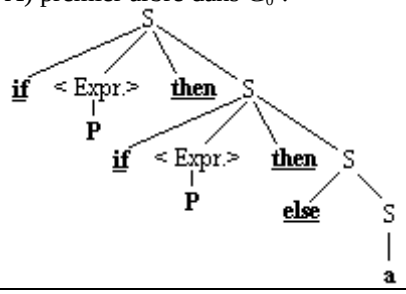
| | | | |
|---|---|---|---|
| 1 : S ⟶ S(SS)<br>2 : S ⟶ ε | 1 : S ⟶ (S)S<br>2 : S ⟶ ε | 1 : S ⟶ S(S)S<br>2 : S ⟶ ε | *Etc..* |

*Ex-2 :* 1°) mot minimal : **( ) )**

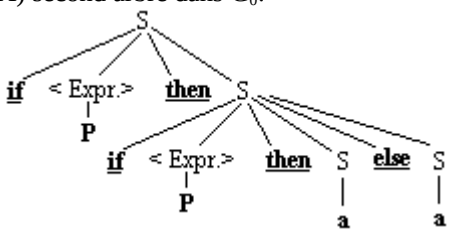
2°) L(G) = { ( $^{n}$ o$^{p}$ )$^{q}$ , n ≥ 1, p ≥ 0, q ≥ 1 }

*Ex-3 :*

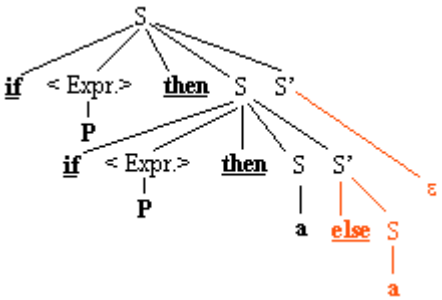
A) premier arbre dans $G_0$ :



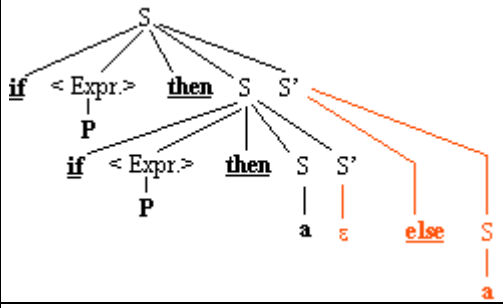

A) second arbre dans $G_0$:



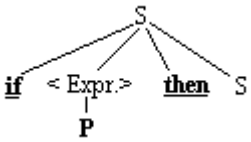

B) premier arbre dans $G_1$ :




B) second arbre dans $G_1$ :

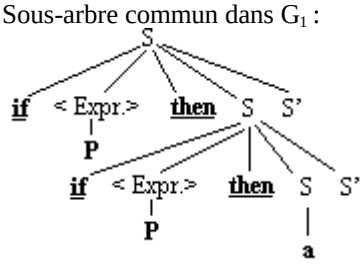



Sous-arbre commun dans $G_0$ :




La grammaire $G_0$ permet une analyse moins profonde que $G_1$ avant que l'ambiguïté se révèle.

Sous-arbre commun dans $G_1$ :




Soit $G_2$

$\mathbf{V_N}$ = {<Expr.> , S}

$\mathbf{V_T}$ = { **if** , **then** , **else** , **P** , **a** , **endif** }

**Axiome** : S

**Règles** 1 : S ⟶ **if** < Expr.> **then** S **endif**

2 : S ⟶ **if** < Expr.> **then** S **else** S **endif**

3 : S ⟶ **a**

4 : < Expr.> ⟶ **P**

On ne peut qu'écrire dans $G_2$ l'une ou l'autre des deux seules phrases distinctes suivantes :

- **if** P **then if** P **then** a **else** a **endif** **endif**
- **if** P **then if** P **then** a **endif** **else** a **endif**

*Ex-4 :* Soit une grammaire $G_2$ répondant à la question

$\mathbf{V_N}$ = {<identif.> , <lettre.> , <chiffre.> , <suite> } , $\mathbf{V_T}$ = { **a, b , … , z , 0 , … , 9** }

**Axiome** : <identif.>

**Règles** 1 : <identif.> ⟶ <lettre.> <suite>

2 : <suite> ⟶ <lettre.> <suite> | < chiffre.> <suite> | ε

3 : <lettre.> ⟶ **a | b | … | z**

4 : < chiffre.> ⟶ **0 | 1 | … | 9**

*Ex-5 :* Appel de la procédure Lire avec le paramètre table : Lire ( table). La **Procedure** Lire( T : tableau ) reçoit un paramètre passé par valeur, donc elle travaille sur une copie du tableau table en saisissant au clavier les données, mais lors de la fin de l'appel le tableau local est détruit et l'original n'a pas été modifié donc le tableau table est resté vide !

*Ex-6 : Somme 1+1/3+1/5+1/7+...+1/19*

| *Boucle for croissante* | *Boucle for décroissante* |
|---|---|
| **program** for_do;<br>**const** max=20;<br>**var**<br>som : real;<br>i : integer;<br>**begin**<br>som:= 0;<br>**for** i := 0 **to** 9 **do**<br>som := som + 1 / (2*i+1);<br>writeln('somme = ', som);<br>**end.** | **program** for_do;<br>**const** max=20;<br>**var**<br>som : real;<br>i : integer;<br>**begin**<br>som:= 0;<br>**for** i := 9 **downto** 0 **do**<br>som := som + 1 / (2*i+1);<br>writeln('somme = ', som);<br>**end.** |

*Ex-7 : chaine → en minuscule*

```
function Lowerstring (ch : string) : string;
var
  i: integer;
  sortie: string;
begin
  sortie := '';
  for i := 1 to length(ch) do
    if ch[i] in ['A'..'Z'] then
          sortie := concat (sortie, chr(ord(ch[i]) + ord('a') - ord('A')))
   else
          sortie := concat(sortie, ch[i] );
   result := sortie
end;
```

*Ex-8 : Tables de vérités*

```
program TableVerite;
var
  a, b, c:boolean;
begin
 writeln('    table du Et :');
 writeln('    a      b      Et');
 writeln('-------------------------');
 for a := false to true do
   for b := false to true do
      writeln( a:7, b:7, a And b:7);
 writeln('**************************');
    writeln('    table du Ou :');
    writeln('    a      b      Ou');
    writeln('-------------------------');
    for a := false to true do
      for b := false to true do
        writeln( a:7, b:7, a or b:7);
    writeln('**************************');
    writeln('    table du Xor :');
    writeln('    a      b      Xor');
    writeln('-------------------------');
    for a := false to true do
      for b := false to true do
         writeln( a:7, b:7, a Xor b:7);
 end.
```

*Ex-9 : implication et équivalence*

| **P => Q = non P ou Q** | **P ⇔ Q = ( P => Q ) et ( Q => P )** |
|---|---|
| **function** implique (p , q : boolean) : boolean;<br>**begin**<br>result := **not** p **or** q<br>**end**; | **function** equivalent (p , q : boolean) : boolean;<br>**begin**<br>result := implique (p,q)**and** implique(q,p)<br>**end**; |