Chapitre 13. Eléments de Turbo-Pascal

Table des matières

1 Les bases en Turbo-Pascal

1.1 Structure d'un programme Pascal.

Un programme est composé d'une **partie déclarative** et d'un **programme principal**.
La partie déclarative permet de définir les variables, fonctions et procédures qu'on utilisera dans le programme principal.

```
Program truc ;                                          }
var x :real ;                                           }
n,i :integer ;                                          }
function f(y :real) :real ;                             }
begin                                                   } Partie déclarative
   f :=...                                              }
end ;                                                   }
procedure fait_ceci(x,y :real ;var t :real) ;           }
begin{contenu de la procédure}end ;                     }
BEGIN
{PROGRAMME PRINCIPAL}
END.
```

1.2 Variables et types.

Une **variable** peut être considérée comme une " case ", munie d'une " adresse ".

Dans la partie déclarative : **déclarer** une variable, c'est réserver cette " case ", en lui donnant un nom, qu'on appelle un identificateur, et en définissant le **type** de cette variable, c'est-à-dire l'ensemble auquel elle appartient.
Un **identificateur** est soit une lettre (`x,u,k...`) soit un mot (`alea, limit...`).
Un **type** est un ensemble muni d'opérations et de fonctions.
Les types utilisés cette année sont les suivants :

a) `integer` : ensemble $\mathbb{Z}$ des entiers relatifs.
Les opérations définies pour ce type sont : $+, *, -$, `div` (`a div b` estle quotient de a par b dans la division euclidienne), `mod` (`a mod b` estle reste dans la division euclidienne de a par b.)
Une variable " indice de boucle" `for..to..do` doit **toujours** être de type `integer`.

b) `longint` : ensemble $\mathbb{Z}$ des entiers relatifs. On utilise ce type quand on doit utiliser des entiers dépassant la valeur 2^{15}. Les opérations sont les mêmes que pour `integer`.

c) `real` : ensemble $\mathbb{R}$ des nombres réels.
Les opérations définies pour ce type sont : $+, *, -, /$ (division dans $\mathbb{R}$).

d) `boolean` : une variable de ce type n'est pas un nombre mais ne peut prendre que deux valeurs, `TRUE` ou `FALSE`.
Remarque : On utilisera également des **expressions booléennes**, pouvant prendre également les 2 valeurs `TRUE` ou `FALSE`, représentant une condition, qui peut être réalisée ou non, souvent sous forme de comparaison.
Exemples : `i=1, f(a)>=0, abs(b-a)<1e-8` (ce qui se traduit par : $|b-a| < 10^{-8}$).
Une expression booléenne, ou une variable booléenne, est attendue après `while` ou `if` ou `until`.

e) `array[a..b] of integer` : Il s'agit d'un tableau unidimensionnel de nombres du type précisé (ici des entiers) dont les "cases" sont numérotées de a à b. Les différentes cases du tableau sont notées `t[k]`, (élément numéro k du tableau) et se comportent comme des variables du type précisé. (Voir chapitre 4.)

Dans le programme principal : on donne une **valeur** à une variable de deux façons possibles :

a) **affectation** : `x :=3 ;` (on affecte la valeur 3 à x).
`x :=y ;` (on affecte à x la valeur contenue à cet instant dans la variable y)
`n :=n+1 ;` (on affecte à n son ancienne valeur augmentée de 1)
`u :=f(u) ;` (on affecte à u l'image de son ancienne valeur par f.
Attention ! ne pas confondre `x :=y ;` (affectation) avec `x=y ;` (comparaison, expression booléenne).

b) **lecture** : `readln(x) ;` (pendant l'exécution du programme, l'utilisateur entre au clavier la valeur qu'il désire affecter à x. L'instruction `readln(x) ;` effectue " l'enregistrement " de cette valeur à l'adresse x.

1.3 Procédures prédéfinies

`write('le nombre cherché est ',a) ;`(affiche le texte écrit entre apostrophes et la **valeur** de a)
`writeln(x) ;` (affiche la valeur de x en allant ensuite à la ligne).
Affichage d'une variable entière : `writeln(k :5)` : aligne à droite le nombre sur un espace de 5 caractères.
Affichage décimal d'un réel : `writeln(x :8 :4)` : aligne à droite le nombre sur un espace de 8 caractères, avec 4 chiffres après la virgule.
`Read(x) ;` (enregistre la valeur donnée au clavier par l'utilisateur et l'affecte à la variable x).
`Readln(x) ;` (même chose en allant à la ligne ensuite).
`Writeln ;` (fait aller à la ligne sans rien afficher).
`Readln ;` (permet d'attendre la commande " entrer " avant de passer à l'instruction suivante) (" arrêt sur image ").

1.4 Boucle `for...to...do.`

Utilisation : Pour effectuer une tâche répétitive, quand le nombre d'étapes est connu.
Ecriture : `for k :=p to n do begin`
`{instructions}`
`end ;`

Remarques :
• quand il n'y a qu'une seule instruction, il est inutile de mettre " begin " et " end ; "
• La variable k est forcément de type entier (integer).
• Les instructions ne doivent jamais modifier la valeur de k (ne pas affecter une valeur à k).
• On doit avoir n $\geq$ p.

Exécution : k prend d'abord la valeur p (affectation initiale) et augmente de 1 à chaque étape. A chaque étape, le programme effectue les instructions encadrées par begin et end. Après l'étape correspondant à k=n, le programme passe à l'instruction qui suit le " end " de la boucle.

Exemples
`u :=a ; for k :=1 to n do u :=f(u) ;` (calcul du terme u_n d'une suite de premier terme $u_0 = a$ et définie par la relation de récurrence $u_{n+1} = f(u_n)$).

`for k := 1 to 20 do writeln(k :3,k*k :6,sqrt(k) :8 :3)) ;` (affiche à l'écran un "tableau de valeurs" contenant les 20 premiers entiers, leurs carrés, leurs racines carrées).

1.5 Boucle `while...do...`

S'emploie quand le nombre d'étapes d'un algorithme n'est pas connu, mais que l'arrêt de cet algorithme est subordonné à la réalisation d'un certain événement : **tant que {condition} faire {instruction}.**

La " condition " est une expression booléenne (qui peut prendre l'une des valeurs " vrai " ou " faux ").

" L'instruction " est généralement composée d'une ou plusieurs affectations . S'il y en a plus d'une il faut les " encadrer " par `begin` et `end ;`.

Exemple : Soit u une suite de premier terme $u_0 = a$ et définie par : $u_{n+1} = f(u_n)$, qui converge vers une limite ℓ. La boucle qui suit calcule le plus petit rang n tel que u_n soit une valeur approchée de ℓ à 10^{-8} près :

```
u :=a ; n :=0 ;
while abs(u-ℓ)>1e-8 do
begin
u :=f(u) ; n :=n+1 ;
end ;
```

1.6 Boucle `repeat..until..`

Même utilisation que la boucle précédente.
Le programme du paragraphe précédent s'écrit alors :

```
u :=a ; n :=0 ;
repeat
      u :=f(u) ;
      n :=n+1 ;
until abs(u-ℓ)<=1e-8
```

On note qu'il n'est pas nécessaire d'encadrer les instructions effectuées dans la boucle par `begin` et `end`.

1.7 Structures conditionnelles

a) Structure conditionnelle simple : `if..then..`

Exemple : `if x>=0 then y :=sqrt(x) ;`
La structure s'écrit : `if <condition> then <instruction> ;`
où la condition est une expression booléenne, et l'instruction est une affectation, ou plusieurs, encadrées alors par `begin` et `end`.
Traduction : si la condition est vraie, alors effectuer l'instruction, sinon, ne rien faire.

b) Structure conditionnelle alternative : `if..then..else`

Exemple : cette fonction nous donne le plus grand de 2 réels :

```
function maximum(a,b :real) :real ;
   begin
      if a>b then maximum :=a
            else maximum :=b ;
   end ;
```

La structure est :

```
   if <condition> then <instruction 1>
                  else <instruction 2> ;
```

Traduction : Si la condition est vraie alors effectuer l'instruction 1, sinon effectuer l'instruction 2.

Remarque : il ne faut jamais mettre un point virgule avant le `else`.

1.8 Tirage au sort

La procédure prédéfinie `randomize ;` permet d'initialiser un tirage aléatoire. On l'inscrira au début de tout programme principal comportant des tirages au sort, c'est-à-dire utilisant la fonction `random ;`, avec ou sans variable.

La fonction `random ;`, sans variable donne un **nombre réel** au hasard dans l'intervalle $[0, 1]$.
Si on a l'instruction `X :=random ;` dans un programme, la variable X est une variable aléatoire qui suit une loi uniforme sur l'intervalle $[0, 1]$ (variable à densité).

La fonction `random(n) ;` donne un **nombre entier** au hasard entre 0 et $n - 1$.
Si on a l'instruction `X :=1+random(100) ;` dans un programme, la variable X est une variable aléatoire qui suit une loi uniforme sur $[\![1, 100]\!]$ (variable discrète).

2 Analyse et Turbo-Pascal

2.1 Déclaration d'une fonction

La déclaration d'une fonction se fait entièrement dans la partie déclarative.
Bien vérifier que le type de la variable d'entrée et celui de la variable de sortie sont cohérents avec le problème proposé.

Exemple 1 :

```
function f(x :real) :real ;
   begin
   f :=2*x/(1+x*x) ;
   end ;
```

Si x est la variable d'entrée, la variable de sortie sera $\dfrac{2x}{1+x^2}$; f est une fonction *réelle* de variable *réelle*.

Dans le programme principal on peut utiliser par exemple `f(5), f(u)` (où u est une variable réelle), `f(k)`(où k est une variable entière). Attention ! ces expressions ne s'utilisent pas comme des variables mais comme des constantes : on peut soit afficher leur valeur, soit affecter leur valeur à une certaine variable, soit les utiliser dans un calcul.

Exemple 2 : lorsqu'il y a un problème d'ensemble de définition, il faut penser à utiliser une structure conditionnelle.

```
   Function f(x :real) :real ;
   begin
   if x<=-1 then f :=0
            else f :=sqrt(x+1) ;
   end ;
```

Exemple 3 : le calcul des valeurs d'une fonction peut s'écrire avec un algorithme plus complexe, avec utilisation éventuelle de variable(s) locale(s).

```
function fact(k :integer) :longint ;
   var i :integer ; x :longint ;
   begin
   if k=0 then fact :=1
         else begin
               x :=1 ;
               for i := 1 to k do x :=x*i ;
               fact :=x ;
               end ;
   end ;
```

2.2 Suites récurrentes

Soit u une suite définie par son premier terme u_0 et la relation : $u_{n+1} = f(u_n)$, où f est une fonction numérique.

2.2.1 Calcul du terme de rang n

La fonction f étant déclarée par ailleurs, n étant un nombre entier donné par l'utilisateur, a étant le premier terme de la suite, on utilise une boucle `for..to..do` et le programme est le suivant :

```
   u :=a ;
   for k :=1 to n do u :=f(u) ;
```

2.2.2 Vitesse de convergence vers une limite connue

On suppose que la limite ℓ est connue, et on cherche le plus petit rang n tel que u_n soit une valeur approchée de la limite, à 10^{-4} près (par exemple), le premier terme étant toujours $u_0 = a$.

```
u :=a ;
n :=0 ;
while abs(u-ℓ)>1e-8 do begin
              u :=f(u) ;
              n :=n+1 ;
              end ;
writeln('le rang cherché est ',n) ;
```

2.2.3 Majoration par une suite géométrique

On suppose que l'on a démontré, à l'aide du théorème des accroissements finis, que pour tout entier n de $\mathbb{N}$:

$$|u_n - \ell| \leq v_0 q^n$$

où q est un réel tel que $-1 \leq q \leq 1$.
Dans ce cas la suite (u_n) converge vers ℓ, et pour calculer cette limite on calcule les termes successifs de la suite tant que le majorant est supérieur à 10^{-4} : quand la boucle s'arrête, la valeur contenue dans la variable u est bien une valeur approchée de ℓ à 10^{-4} près.

```
u :=a ; v :=v0 ;
while v>1e-4 do begin
         u :=f(u) ;
         v :=q*v ;
         end ;
writeln('valeur de la limite : ',u :7 :4) ;
```

2.2.4 Récurrence portant sur plusieurs termes

Exemple : Soit u la suite définie par $u_0 = a$, $u_1 = b$, et la relation : $\forall n \in \mathbb{N} \quad u_{n+2} = \sqrt{u_n u_{n+1}}$.
L'idée, pour calculer le terme de rang n donné, est de "faire tourner" les termes successifs de la suite sur 3 variables notées u,v,w.

```
u :=a ; v :=b ;
for k :=2 to n do begin
            w :=sqrt(u*v) ;
            u :=v ;
            v :=w ;
            end ;
writeln('le terme de rang ',n,' est ',w :6 :4) ;      {pour  n ≥ 2}
```

2.3 Calcul de sommes et applications.

2.3.1 Généralités

Soit à calculer la somme $S_n = \sum_{k=1}^{n} f(k)$, où f est une certaine fonction de variable réelle ou entière.
Dans la partie déclarative, on déclare la fonction f ainsi qu'une variable s qui va recevoir les valeurs successives de S_n.
Dans le programme principal, on initialise la somme à 0, et à chaque étape d'une boucle `for..to..do` on ajoute le terme de rang k à la somme :

```
s :=0 ;
for k :=1 to n do s :=s+f(k) ;
```

Remarque : dans les cas simples il n'est pas nécessaires de déclarer la fonction.
Exemple : pour calculer la somme des carrés des n premiers entiers non nuls :

```
   s :=0 ;
   for k :=1 to n do s :=s+k*k ;
```

2.3.2 Calcul approché de la somme d'une série

Si la série de terme général u_n est convergente, sa somme est :

$$S = \lim_{n\to+\infty} S_n = \lim_{n\to+\infty} \sum_{k=1}^{n} u_k$$

Pour n "grand", la somme partielle S_n sera une valeur approchée de la somme de la série.

Cas particulier : série alternée. Soit (u_n) une suite positive décroissante de limite nulle, et (v_n) la suite définie par : $\forall n \in \mathbb{N} \quad v_n = (-1)^n u_n$.
Alors la série de terme général v_n est convergente.
Soit S_n la somme partielle de cette série, alors pour tout entier n impair, $S_n \leq S \leq S_{n+1}$. D'autre part comme $\lim_{n\to+\infty} (S_{n+1} - S_n) = \lim_{n\to+\infty} u_{n+1} = 0$, la somme partielle S_n (ou S_{n+1}) est une valeur approchée de S à 10^{-4} près dès que $|u_{n+1}| < 10^{-4}$.

Exemple : calcul de la somme de la série harmonique alternée.

```
Program Harmalt ;
var n :integer ;
      s :real ;
BEGIN
   s :=0 ; n :=1 ;
   while 1/(n+1)>1e-4 do begin
                  if (n div 2)=0 then u :=u + 1/n else u :=u - 1/n ;
                  n :=n+1 ;
                  end ;
   writeln('la somme de la série est : ',u :7 :4) ;
   readln ;
END.
```

2.3.3 Calcul approché d'intégrales.

On utilise le **théorème des sommes de Riemann**, ce qui revient à appliquer la **méthode des rectangles** :
Si f est une fonction continue sur $[a, b]$, alors :

$$\int_a^b f(t)dt = \lim_{n\to+\infty} \frac{b-a}{n} \sum_{k=0}^{n-1} f\left(a + k\frac{b-a}{n}\right) = \lim_{n\to+\infty} \frac{b-a}{n} \sum_{k=1}^{n} f\left(a + k\frac{b-a}{n}\right)$$

Chacune des deux sommes S_n et S'_n de cette formule est une valeur approchée de l'intégrale, pour n "assez grand".
Leur moyenne est également une valeur approchée, encore plus précise (méthode des trapèzes).

Calcul de l'intégrale sur $[0, 1]$ de la fonction : $f(x) = \dfrac{x \ln x}{x-1}$.

Cette fonction admet un prolongement par continuité sur $[0, 1]$, en prenant : $f(0) = 0$ et $f(1) = 1$.

```
Program calculintegrale ;
Var s,t :real ;
      k,n :integer ;
function f(x :real) :real ;
      begin
```

```
    if x=0 then f :=0 else if x=1 then f :=1 else f :=x*ln(x)/(x-1) ;
    end ;

BEGIN
    s :=0 ; n :=100 ;
    for k :=0 to n-1 do s :=s+f(k/n) ;
    s :=s/n ;
    t :=s - f(0)/n + f(1)/n ;
    writeln('v.a. par méthode des rectangles :',s :8 :6,' ou ',t :8 :6) ;
    writeln('v.a. par méthode des trapèzes :',(s+t)/2 :8 :6) ;
    readln ;
END.
```

2.4 Solution d'une équation

On suppose qu'une fonction f est continue et strictement monotone sur un intervalle $[a, b]$, et que 0 appartient à l'intervalle image de $[a, b]$ par$f$. Alors, d'après le théorème de la bijection, l'équation $f(x) = 0$ admet une unique solution α dans l'intervalle $[a, b]$.
On cherche à déterminar une valeur approchée de α à 10^{-4} près (par exemple).

2.4.1 Méthode de dichotomie

On part de l'idée qu'une fonction monotone *change de signe* dans l'intervalle où elle s'annule. On coupe l'intervalle en 2, puis encore en 2, etc, en testant à chaque étape dans quel demi-intervalle se trouve la racine. Lorsque la longueur de l'intervelle $[a, b]$ est inférieure à 10^{-4} (par exemple), les deux nombres a et b sont des valeurs approchées à 10^{-4} près de la racine α.

```
While b-a>1e-4 do begin
              c :=(a+b)/2 ;
              if f(a)*f(c)<0 then b :=c else a :=c ;
              end ;
```

2.4.2 Méthode de Newton

On suppose que la dérivée de la fonction f est bornée et ne s'annule pas sur $[a, b]$.
On considère la tangente à la courbe de f au point d'abscisse a : cette droite coupe l'axe des abscisses en un point d'abscisse x_1.
Cacul de x_1 : L'équation de la tangente est : $y - f(a) = f'(a)(x - a)$.
Cette droite contient le point $(x_1, 0)$ donc : $-f(a) = f'(a)(x_1 - a)$. On en déduit :

$$x_1 = a - \frac{f(a)}{f'(a)}$$

De même la tangente à la courbe de f au point d'abscisse x_1 coupe l'axe des abscisses en un point d'abscisse $x_2 = x_1 - \dfrac{f(x_1)}{f'(x_1)}$.
En renouvelant l'opération, on définit une suite $(x_n)_{n\in\mathbb{N}}$, en prenant $x_0 = a$, et pour tout entier naturel n, $x_{n+1} = x_n - \dfrac{f(x_n)}{f'(x_n)}$.
Cette suite converge vers la racine α de l'équation $f(x) = 0$.
Il suffit d'écrire un programme calculant le terme de rang n d'une suite définie par récurrence, pour n "assez grand". (Voir paragraphe 10.1)

3 Procédures.

3.1 Généralités

3.1.1 Déclaration de la procédure : passage par valeur ou par variable.

Une procédure est un sous-programme entièrement écrit dans la partie déclarative. Ce programme effectue une certaine " tâche " lorsque l'on effectue dans le programme principal un appel de la procédure utilisant l'identificateur (= nom) de la procédure ainsi que la ou les variable(s) auxquelles on veut l'appliquer. On peut ainsi appeler plusieurs fois une même procédure, en l'appliquant éventuellement à différentes variables, ce qui a pour avantage d'alléger et clarifier le programme principal.

a) Passage par variable (ou par adresse)

```
procedure truc(var x :real) ;
begin
(...) {un calcul sur la variable x, éventuellement sous forme d'une boucle...}
end ;
```

Dans ce cas, si on effectue dans le programme principal l'appel : `truc(u) ;`, la procédure effectue le calcul contenu dans la procédure " truc " en remplaçant x par la variable u, et la modification sur u opérée par cette procédure est conservée dans le programme principal.

b) Passage par valeur

```
procedure truc(x :real) ;
begin
(...) {un calcul utilisant la variable x}
end ;
```

Dans ce cas, si on effectue dans le programme principal l'appel : `truc(u) ;`,la procédure utilise une copie de la variable u pour effectuer le calcul, mais au retour dans le programme principal la valeur de u n'est pas modifiée.

c) Procédure sans variable

```
procedure truc ;
begin
(...)
end ;
```

Cette procédure peut éventuellement utiliser les variables du programme principal. Il faut alors faire attention aux valeurs prises par ces variables au moment de l'appel de la procédure.

Conclusion : on utilise le passage par variable quand on veut que la procédure modifie la valeur de la variable (exemples : procédure " échange ", procédure de lecture d'une variable dont la valeur est entrée au clavier,...), on utilise le passage par valeur quand on veut simplement utiliser la valeur d'une variable, sans la modifier (exemples : procédure d'affichage d'un résultat, procédure de calcul d'un discriminant à partir des paramètres a, b, c d'une équation du second degré...).
Remarque : Toute variable du programme principal utilisée dans une procédure (avec ou sans variable) passe par variable dans la procédure.

3.1.2 Variable locale

Il s'agit d'une variable interne au sous-programme défini par la procédure et qui ne sert pas en dehors.
Exemple : un indice de boucle `for..to..do..` pour une boucle interne à la procédure.

3.2 Exemples de procédures

3.2.1 Calcul du terme de rang n d'une suite définie par récurrence

```
Program chap3ex1 ;
var u : real ; n :integer ;
function f(x :real) :real ;
      (...)
procedure suite(n :integer) ;
var k : integer ;
   begin
   for k :=1 to n do u :=f(u) ;
   end ;
```

```
BEGIN
writeln('donner le premier terme u0 de la suite') ;
readln(u) ;
writeln('donner le rang') ;
readln(n) ;
suite(n) ;
writeln('le terme de rang ',n,' est ',u :8 :6) ;
readln ;
END.
```

Remarques : Ici le nombre entier n est passé par valeur, car on n'a pas à le modifier, mais simplement l'utiliser.
La variable u n'est pas une variable de la procédure, mais du programme principal : elle passe automatiquement par variable dans la procédure, et la modification effectuée sur u par la procédure est effective dans le programme principal.
La variable k est une variable locale de la procédure, on n'en a pas besoin dans le programme principal.

3.2.2 Recherche de la solution d'une équation par dichotomie

Il s'agit d'afficher une valeur approchée à 10^{-4} près de la solution de l'équation $f(x) = 0$ dans l'intervalle $[1; 3]$:

```
program chap3ex2 ;
var a,b :real ;
function f(x :real) :real ;
      (...)
procedure dichotomie(var a,b :real) ;
var c : real ;
   begin
   while b-a>1e-4 do begin
               c :=(a+b)/2 ;
               if f(a)*f(c)<0 then b :=c else a :=c ;
               end ;
BEGIN
a :=1 ; b :=3 ;
dichotomie(a,b) ;
writeln('la solution est : ',a :6 :4) ;
readln ;
END.
```

3.2.3 Procédure échange

Cette procédure est très utilisée pour manipuler des tableaux de nombres, en particulier dans les algorithmes de tri (voir chapitre 5).
Il s'agit d'échanger les valeurs de deux variables du même type, notée pour l'instant x et y.
Si on effectue `x :=y ;` la valeur de x est remplacée par celle de y, et on perd la valeur de x. C'est pourquoi on utilise une variable auxiliaire z, qui va recevoir provisoirement la valeur de x avant de la remplacer par y.

```
procedure echange(var x,y :integer) ;
   var z :integer ;
   begin
   z :=x ; x :=y ; y :=z ;
   end ;
```

Remarques : Le type est `integer` dans cet exemple, la variable locale z est donc de ce même type.
Les deux variables x et y sont passées par variable, puisque la procédure doit entraîner une modification de ces variables.

4 Récursivité.

4.1 Fonction récursive

Une fonction est dite récursive dans le cas où elle s'applique aux entiers naturels, et que la valeur au rang $n+1$ dépend directement de la valeur au rang n.
Dans ce cas, dans la déclaration de la fonction on peut appeler cette même fonction, comme s'il s'agissait d'une boucle.

4.1.1 Fonction factorielle

On sait que cette fonction est définie sur $\mathbb{N}$ par :

$$0! = 1 \quad \text{et} \quad \forall n \in \mathbb{N}^* \quad n! = n(n-1)!$$

La fonction définissant $n!$ en Turbo-Pascal (forme récursive) est donc :

```
function fact(n :integer) :longint ;
   begin
   if n=0 then fact :=1 else fact :=n*fact(n-1) ;
   end ;
```

Remarque : la variable de sortie de cette fonction est `longint ;` ou même `real ;` car on atteint très vite des grandes valeurs entières, dépassant la capacité du type `integer ;`.

4.1.2 Fonction puissance

On considère la fonction puissance comme une fonction de 2 variables : $f(x,n) = x^n$.
Pour un réel x fixé, on a alors la définition par récurrence :

$$x^0 = 1 \quad \text{et} \quad \forall n \in \mathbb{N}^* \quad x^n = xx^{n-1}$$

Le programme récursif définissant la fonction puissance est donc :

```
function f(x :real ;n :integer) :real ;
   begin
   if n=0 then f :=1 else f :=x*f(x,n-1) ;
   end ;
```

4.2 Procédure récursive

On peut appliquer la récursivité à toute tâche dépendant d'un entier naturel, tel que le résultat au rang n soit directement dépendant du résultat au rang $n-1$.
Exemple : soit à afficher le terme de rang n de la suite définie par récurrence : $u_0 = a$ et pour tout entier n non nul, $u_n = f(u_{n-1})$.

```
procedure suite(u :real ; n :integer) ;
   begin
   if n=0 then writeln(u) else begin suite(f(u),n-1) ;
   end ;
```

Remarque : si on veut passer la variable u par adresse, on est obligé de définir une variable locale v car la procédure ne peut pas marcher avec $f(u)$ qui est considérée comme une **constante** et non comme une variable. On contourne ce problème en donnant à une variable la valeur de cette constante.

D'autres procédures récursives seront abordées dans les chapitres suivants.

5 Tableaux, compteur, tris.

5.1 Tableau.

5.1.1 Généralités

C'est un type de variable composée, qui permet de manipuler simultanément un nombre quelconque de variables de même type.
Dans la partie déclarative : `var t :array[1..200] of integer ;` {permet de " réserver " 200 " cases " numérotées destinées à recevoir des valeurs entières}.
Remarque : pour utiliser un tableau t dans une procédure, il faut que la variable t soit de type simple. Comment faire ? Tout simplement en définissant un nouveau type, à partir du type tableau, qui sera alors considéré comme un type simple.
On écrit dans la partie déclarative, juste après le titre du programme :

```
type tab=array[1..200] of integer ;
var t :tab ;
```

Le type `tab` s'emploie alors de la même façon que n'importe quelle variable simple (entière ou réelle) mais avec les caractéristiques d'un tableau, c'est-à-dire que l'on ne peut l'utiliser qu'à l'aide d'une boucle.

Dans le programme principal : un tableau s'utilise toujours avec une boucle, le cas le plus fréquent étant celui d'une boucle `for...to...do.`

Exemple :

```
for i := 1 to 200 do begin
                  t[i] := random(1000) ;
                  write(t[i] :4) ;
                  end ;
```

{tire au sort un tableau de 200 entiers compris entre 0 et 999, et l'affiche}.

Remarque : i est le numéro de la "case", t[i] est la valeur contenue dans la "case".

5.1.2 Compteur

Il s'agit d'un entier, noté par exemple n, que l'on initialise à la valeur 0, et auquel, à chaque étape d'une certaine boucle, on va ajouter 1 si une certaine condition est remplie.

Exemple 1 : pour compter le nombre d'éléments d'un tableau t qui sont supérieurs à 700 :

```
For i := 1 to 200 do if t[i]>700 then n :=n+1 ;
Writeln('le nombre cherché est ',n) ;
```

On peut aussi utiliser un compteur pour compter le nombre de passages dans une boucle, le nombre d'affectations effectuées, le nombre de comparaisons...

Exemple 2 :soit une suite définie par son premier terme u_0, et par une relation de récurrence du type : $u_{n+1} = f(u_n)$, et dont on connaît par ailleurs la limite l.

```
n :=0 ;
While abs(u-l)>1e-8 do begin
                  u :=f(u) ;
                  n :=n+1 ;
                  end ;
```

Le nombre n obtenu est le plus petit rang pour lequel u_n est une approximation de l à moins de 10^{-8} près.

5.2 Tri d'un tableau de valeurs aléatoires.

Dans la partie déclarative :

```
type tab=array[1..100] of integer ;
var t :tab ;i,j :integer ;
procedure echange(var x,y :integer) ;
var z :integer ;
begin
z :=x ; x :=y ; y :=z ;
```

```
end ;
```

Dans le programme principal :
```
randomize ;
for i :=1 to 100 do t[i] :=random(1000) ;
```
(On remplit un tableau de 100 cases avec des nombres au hasard entre 0 et 999.)

5.2.1 Tri à bulles

Cet algorithme a pour but de classer les éléments d'un tableau T de n entiers dans un ordre croissant (on pourrait l'adapter pour les trier dans l'ordre décroissant...).

```
for j :=n-1 downto 1 do
   for i :=1 to j do if T[i]>T[i+1] then echange(T[i],T[i+1]) ;
```

Remarque : La boucle intérieure appelée **petit tri** a pour effet de placer le plus grand des $j+1$ premiers éléments du tableau en $(j+1)$-ème position.

5.2.2 Recherche dans un tableau trié.

On suppose qu'on a tiré au sort les valeurs figurant dans un tableau de n entiers, et on pose $x = \mathrm{T}[1]$ (ou tout autre valeur du tableau, que l'on veut particulariser.). On effectue le tri à bulles. Les deux algorithmes suivants sont des fonctions qui donnent le rang de l'élément x dans le tableau une fois trié :

a) Recherche linéaire, ou séquentielle :
```
function rang(x :integer) :integer ;
   Var i :integer ;
   begin
   i :=1 ;
   while t[i]<x do i :=i+1 ;
   rang :=i ;
   end ;
```

b) Recherche dichotomique :

```
function rang(x, prem, der :integer) :integer ;
   var med :integer ;
   begin
   med :=(prem+der)div 2 ;
   if T[med]=x then rang :=med
            else if T[med]>x then rang :=rang(x,prem,med)
                     else rang :=rang(x,med,der) ;
   end ;
```

Remarque : La recherche dichotomique est une fonction **récursive**. Elle est plus rapide que la recherche séquentielle pour les tableaux de taille importante.

6 Simulation d'expériences aléatoires.

On sait que les fonctions `random` et `random(n)` permettent d'effectuer des tirages au sort. On va donc utiliser ces fonctions pour simuler des expériences aléatoires permettant d'obtenir une valeur pour une variable suivant une certaine loi de probabilité, usuelle ou non. En effet cette variable aléatoire est souvent définie au départ comme nombre de ceci, ou rang de cela, à l'issue d'un certain processus.

6.1 Loi géométrique

Une variable X suit une loi géométrique de paramètre p si X est le rang du premier succès, au cours d'une suite de tirages indépendants tels qu'à chaque tirage la probabilité du succès est p.
Supposons qu'on cherche à simuler la variable X avec $p = \frac{2}{5}$. Il faut donc simuler un événement de probabilité $p = \frac{2}{5}$.
Or si `Y=random(5)` la variable Y suit une loi *uniforme* sur $[\![0, 4]\!]$. Donc $P(Y \leq 1) = \frac{2}{5}$.
Il suffit de renouveler l'expérience `Y=random(5)` jusqu'à ce que Y prenne une valeur 0 ou 1. la valeur de X est alors celle du *compteur* qui compte les tirages successifs.

```
BEGIN
   randomize ;
   X :=0 ;
   repeat
   Y :=random(5) ;
   X :=X+1 ;
   until Y<=1 ;
   writeln('X=',X) ;
END.
```

Remarque : On peut remplacer $\frac{2}{5}$ par un paramètre p quelconque, sachant que la fonction `random` suit une loi uniforme sur $[0, 1]$, et donc que si `Y :=random` on a $P(Y \leq p) = p$. Cela donne le programme suivant :

```
BEGIN
   randomize ;
   writeln('donner le paramètre') ;
   readln(p) ;
   X :=0 ;
   repeat
   Y :=random ;
   X :=X+1 ;
   until Y<=p ;
   writeln('X=',X) ;
END.
```

Si on renouvelle cette expérience un grand nombre de fois, par exemple 10 000 fois, la moyenne des valeurs obtenues pour X est une estimation de l'espérance de X.
On peut ainsi vérifier (à l'aide d'une boucle `for..to..do`) que l'espérance d'une variable de loi géométrique $\mathcal{G}(p)$ est $\frac{1}{p}$.

6.2 Loi binomiale

Une variable X suit une loi binomiale de paramètres n et p si X est le nombre de succès, sur n tirages indépendants, la probabilité du succès à chaque tirage étant p.
Pour simuler X on va donc utiliser une boucle `for..to..do` : à chaque étape de la boucle on effectue un tirage, et on compte le nombre de succès :

```
BEGIN
```

```
   randomize ;
   writeln('donner les paramètres n et p') ;
   readln(n) ; readln(p) ;
   X :=0 ;
   for k :=1 to n do
         begin
         Y :=random ;
         if Y<=p then X :=X+1 ;
         end ;
   writeln('X=',X) ;
END.
```

6.3 Loi hypergéométrique

Il s'agit cette fois de modéliser un tirage de k numéros parmi N, *sans remise*.
Prenons pour simplifier $N = 100$, et $p = 0,35$ (sur les 100 numéros, 35 sont associés au "succès").
On définit alors un tableau de 100 nombres, et on le remplit avec 35 valeurs 1 et 65 valeurs 0. Si on tire une "case" au hasard dans ce tableau, c'est-à-dire un numéro entre 1 et 100, la probabilité que sa valeur soit 1 est bien 0,35.
Pour simuler le tirage sans remise, après le premier tirage on échange la valeur du numéro tiré avec celui du dernier numéro `t[100]`, et on effectue le tirage suivant sur les 99 premiers numéros. La valeur du numéro tiré est échangée avec `t[99]`, et on fait le tirage suivant sur les 98 premiers numéros.
On a ainsi mis de côté les numéros tirés, après chaque tirage.
Bien entendu, la valeur de la variable X est celui d'un compteur qui compte le nombre de 1 tirés au cours de ce processus :

```
Program loihyper ;
var t : array[1..100] of integer ;
   i,k :integer ;
procedure echange (...)
BEGIN
   readln(k) ;
   for i :=1 to 35 do t[i] :=1 ;
   for i :=36 to 100 do t[i] :=0 ;
   X :=0 ;
   for i :=1 to k do
      begin
      a :=1+random(101-i) ;
      if t[a]=1 then X :=X+1 ;
      echange(t[a],t[101-i]) ;
      end ;
   writeln('X=',X) ;
END.
```

6.4 Loi uniforme sur un intervalle $[a, b]$

Comme la fonction `random` donne un réel au hasard compris entre 0 et 1, le produit `c*random` donne un réel au hasard entre 0 et c.
Par conséquent pour obtenir un réel au hasard entre a et b, on écrit la fonction suivante :

```
function X(a,b :real) :real ;
begin
X :=a+(b-a)*random ;
end ;
```

6.5 Loi exponentielle

Soit $X \hookrightarrow \mathcal{U}(]0,1])$, et λ un nombre réel positif.
On pose : $g(x) = -\dfrac{\ln x}{\lambda}$. La fonction g définit une bijection de $]0,1]$ vers $\mathbb{R}_+$. Soit $Y = g(X)$. La fonction de répartition de Y est donnée par :
• si $x < 0$, $\quad F_Y(x) = 0$
• si $x \geq 0$, $\quad F_Y(x) = P(-\dfrac{\ln X}{\lambda} \leq x) = P(X \geq e^{-\lambda x}) = 1 - e^{-\lambda x}$
Y suit donc une loi exponentielle de paramètre λ.

La fonction suivante donne une valeur de $Y \hookrightarrow \mathcal{E}(a)$, où a est un réel strictement positif.

```
function Y(a :real) :real ;
var X :real ;
begin
X :=random ;
Y :=-ln(X)/a ;
end ;
```

Remarque : cette méthode peut s'appliquer à la construction de n'importe quelle variable $Y = g(X)$, où $X \hookrightarrow \mathcal{U}(]0,1[)$ et où g est une bijection de $]0,1[$ vers l'ensemble de valeurs de Y, avec :
`Y :=g(random) ;`
la fonction g étant déclarée par ailleurs.

6.6 Loi normale centrée réduite

On utilise le théorème de la limite centrée :
Si $(X_n)_{n\in\mathbb{N}^*}$ est une suite de variables aléatoires indépendantes et de même loi, $\overline{X_n} = \dfrac{1}{n}\displaystyle\sum_{k=1}^{n} X_k$,
$Z_n = \dfrac{\overline{X_n} - E(\overline{X_n})}{\sigma(\overline{X_n})}$, alors la suite $(Z_n)_{n\in\mathbb{N}^*}$ converge en loi vers une variable aléatoire de loi normale centrée réduite.

On va prendre une suite de variables aléatoires de loi uniforme sur $[0,1]$ (la plus facile à modéliser en Turbo-Pascal, à l'aide de la fonction `random`!) et caculer une valeur de Z_n pour n assez grand.
La fonction suivante donne une valeur de la variable T, qui suit une loi normale centrée réduite, pour un n donné dans le programme principal par l'utilisateur.

```
function Normale(n :integer) ;
   var k :integer ; s :real ;
   begin
   s :=0 ;
   for k :=1 to n do s :=s+random ;
   s :=s/n ;
   Normale :=sqrt(12*n)*(s-0.5) ;
   end ;
```