Algorithmique et programmation Pascal

Emilie MORVANT

milie.morvant@gmail.com

Saint-Louis

Prépa ECE 1

Année 2008-2009

LaTeX

Table des matières

Première partie
Cours

1 Introduction

1.1 Quelques mots sur l'algorithmique

De nos jours, l'algorithmique est associée à la programmation informatique. Cependant, elle ne date pas d'hier puisque les premiers algorithmes remontent à environ 1800 ans avant J.C avec les babyloniens, ensuite Euclide (PGCD) et beaucoup d'autres. Contrairement à ce que l'on pourrait penser, les algorithmes ne se traitent pas qu'avec des nombres ; il en existe énormément qui traitent d'autres données, comme l'algorithme génétique (ADN), les algorithmes de sortie d'un labyrinthe, les algorithmes de jeux, ...
Les algorithmes ne se décrivent pas avec un langage de programmation contrairement aux idées reçues et donc ne nécessitent pas un ordinateur pour les écrire. Nous allons donc apprendre à résoudre des problèmes par le biais d'algorithmes et ensuite à les appliquer en deux étapes :

- Ecriture d'un algorithme c'est à dire une méthode permettant de trouver une solution à partir des données d'un problème.
- Ecriture d'un programme qui consiste à traduire un algorithme pour une machine dans un langage de programmation donné, ici le `Pascal`.

Définition 1.1. *Un algorithme est une description finie d'un calcul qui associe un résultat à des données. Il est composé de 3 parties :*

- **son nom**
- **sa spécification** *qui décrit quels sont les paramètres en entrée et quel est le résultat en sortie. Elle décrit le problème résolu par l'algorithme (la fonction résolu par l'algorithme).*
- **son corps** *qui décrit la démarche de résolution d'un problème dans un langage algorithmique, il fournit divers objets et instructions primitives ainsi que des moyens de les composer, mais ne nous empêche pas de faire appel à un algorithme dans un autre.*

Remarque : Par le terme *langage algorithmique* il ne faut pas entendre quelque chose de normé mais d'évolutif car la syntaxe est propre à l'auteur, mais si l'on fonctionne de cette manière, il y a de forte chance de ne pas se faire comprendre par les autres d'où la nécessité d'utiliser les mêmes notations par pure lisibilitée pour les autres.

Exemple : Un exemple d'algorithme permettant de définir si un entier a est pair :

Algorithme 1: estPair

Données : $a \in \mathbb{N}$
Résultat : VRAI si a est pair, FAUX sinon
début
si a *modulo* $2 = 0$ **alors**
  renvoyer VRAI
sinon
  renvoyer FAUX
  fsi
fin

Explications : ici modulo sert à renvoyer le reste de la division euclidienne de a par 2. La division euclidienne de a par b s'écrit d'une manière unique $a = b \times q + r$ avec q le quotient et r le reste tels que $0 \leq r < b$.
Une fois un algorithme écrit, on l'utilise par application à des arguments.

Exemple : Fonctionnement de l'algorithme précédent avec l'exécution de `estPair(21)` :
- Substituer 21 à a dans le corps de `estPair`
- Si $(21\ mod\ 2) = 0$ alors on va renvoyer $true$
- Sinon renvoyer $false$.

Le résultat de cette exécution est $false$.

Par analogie avec les mathématiques, un algorithme est semblable a une fonction f, les objets x sur lesquels agit f portent là aussi le nom de variables, et leurs images $y = f(x)$ car elles peuvent servir de nouvelles variables sur lesquelles faire agir une autre fonction g. Dans un souci de simplification, on admet généralement que les constantes sont des cas particuliers de variables.
Plus précisément, une variable est une donnée désignée par un nom précis et immuable, mais dont la valeur est susceptible de changement au cours du déroulement de l'algorithme.
Reprenons l'analogie avec les mathématiques : une fonction f définie sur un ensemble E, est à valeurs dans un ensemble F très souvent distinct de E.
L'algorithmique fait un grand usage du cas où F ne possède que deux éléments, appelés selon le contexte 0 et 1, **oui** et **non**, **vrai** ou **faux**. On appelle alors **variables booléennes**, ou **indicateurs booléens** les variables dont les images (leurs valeurs) sont nécessairement **vrai** ou **faux**.
Dans la pratique, il s'agit toujours de propositions logiques simples, souvent dépendant d'un ou plusieurs paramètres, auxquels les mathématiques donnent une valeur de vérité : ainsi la proposition *"a est un nombre pair"* est vraie si $a = 3$ et fausse si $a = 4$.

Définition 1.2. *En informatique, il faut différencier deux types de variables :*
- **paramètre formel** *que l'on nommera dans la suite du cours* **paramètre** *: il s'agit de la variable utilisée dans le corps de l'algorithme (par ex : si on avait déclaré une variable dans le corps de l'algorithme* `estPair` *elle serait un paramètre formel).*
- **paramètre effectif** *que l'on nommera dans la suite du cours* **argument** *: il s'agit de la variable (ou valeur) fournie lors de l'appel d'un algorithme (par ex : dans l'algorithme* `estPair` *a en est un car c'est une valeur donnée à l'algorithme lors de son appel pour savoir si elle est pair ou non).*
  On obtient sa valeur en substituant dans le corps de l'algorithme les arguments (par exemple 21) aux paramètres de l'algorithme (ici a) et en appliquant le corps substitué de l'algorithme obtenu dans l'étape précédente ; la valeur résultat est celle donnée par l'instruction renvoyer.

1.2 Quelques mots sur le langage `Pascal`

Le langage de programmation `Pascal`[1] a été inventé par *Niklaus Wirth* dans les années 1970. C'est un langage de programmation dit impératif. Il a été conçu pour servir à l'enseignement de la programmation de manière rigoureuse mais simple, il se caractérise par une syntaxe claire et facilitant la structuration des programmes. C'est pourquoi, il vous sera demander **d'écrire les différents algorithmes en suivant la syntaxe de ce langage**.
Aujourd'hui, lorsque l'on parle du langage `Pascal`, on l'associe au `Turbo Pascal`, créé par *Borland*

[1] dont le nom vient du mathématicien français Blaise Pascal

en 1986. Mais attention, ce dernier n'est pas un langage de programmation, c'est un environnement de dévolopement intégré[2] assez complet pour le langage `Pascal`.

[2]un EDI est un programme regroupant un éditeur de texte, un compilateur, des outils automatiques de fabrication, et souvent un débogueur.

2 Structure d'un programme en Pascal

1. **Déclaration du programme**.
   C'est l'entête du programme. Sa syntaxe est :

   ```
   PROGRAM nom;
   ```

2. **Déclaration des objets**.
   On déclare tous les objets utilisés dans le programme : constantes puis variables.
   Sa syntaxe est :

   ```
   CONST  a=2;
          vrai=TRUE;
   VAR    i,k,n : integers                  {i,k,n sont de types entier}
          x,a,b : real                      {x,a,b de type réel}
          test : boolean                    {variable de type booléenne : }
                                            {  sa valeur est true ou false}
          liste : array [1..10] of integer  {variable de type tableau}
   ```

   Remarque : Avant la déclaration des objets, si l'on utilise une bibliothèque particulière, il faut l'indiquer avec la syntaxe suivante : `uses la_bibliothèque ;`.
   Les bibliothèques sont différentes en fonction de l'environnement utilisé et permettent d'utiliser des fonctions particulières.
   Par exemple, afin de pouvoir effacer l'écran sous l'environnement `DOS` on inclut dans le programme la bibliothèque `crt`. La fonction a utilisée dans le programme pour effacer l'écran est alors : `clrscr ;`.

3. **Définition des procédures et fonctions**.
   Les fonctions et procédures seront vues ultérieurment.
   La structure d'un programme simple est : 1.2.4. ; celle d'un programme structuré comporte cette 3° partie.

4. **Programme principal**.
   C'est le corps du programme : on liste les instructions à exécuter.
   Sa syntaxe est :

   ```
   BEGIN instructions END.
   ```

Sauf mention contraire, chaque ligne de commande et instruction se finit par un **;**.
Afin de faciliter la lisibilité du programme, on peut y ajouter des **commentaires** pour expliquer certaines instructions ou nom de variable. Ces commentaires peuvent être insérées n'importe où du moment qu'ils sont placés entre accolades.

A noter que que le code source brut d'un programme `Pascal` peut porter les extensions[3] `.pas`, le plus courant, `.p` ou encore `.pp`.

[3] L'extension d'un fichier est ce qui permet d'identifier son format, elle se situe après un '.' : par exemple `image.jpg`, `.jpg` indentifie un fichier de type image de format jpeg dont le nom est `image` ou alors `texte.txt` qui correspond à un fichier texte donc le nom est `texte`.

3 Objets et actions élémentaires

3.1 Les objets

Les objets sont des données constantes ou variables :

- une donnée **constante** est une donnée dont la valeur est **fixée** par le programmeur et qui **reste inchangée** au cours de l'exécution du programme.
  Un exemple important est la constante "chaîne de caractères" (ou `string`) : `'bonjour'` ou `'rerevgeetr'` ... Pour définir une chaîne de caractères, il suffit de mettre une suite caractères à l'intérieur de ' '.
- une donnée **variable** est une donnée dont la valeur est **initialisée** par le programmeur ; cette valeur **peut-être modifiée** au cours de l'exécution du programme.

Tous ces objets vont avoir des noms définis par le programmeur : ces noms s'appellent des **identificateurs**. Ces derniers ne peuvent pas être des mots réservés, i.e. des mots dont la signification est prédéfinie par le logiciel, tels que `REAL`, `BEGIN`, ...
Il y a quelques règles supplémentaires : il ne peut y avoir ni espaces, ni lettres accentuées dans un identificateur, et le premier caractère ne doit pas être un chiffre.
Le compilateur `Pascal` ne distinguant pas les majuscules des minuscules, les identificateurs `IDEM`, `Idem`, `iDEm`, `idem` sont confondus.
Chaque identificateur devra être déclaré en **2.** pour être compris.

3.1.1 Principaux types de données

- type entier : `integer`
  Ils doivent être compris entre -32768 et +32767.
- type réel : `real`
- type booléen : `boolean`
  Il définit deux valeurs logiques : vrai ou faux.
  Exemple : Si x est connu, la variable $(x > 0)$ est de type booléen.
- type tableau : `array`
  Il faut alors préciser le type des éléments du tableau qu'il doit prendre en compte. On verra ce type plus en détail dans une prochaine partie.

Cette liste correspond aux types qui nous serons utiles dans ce cours, à noter qu'il existe les types caractères (`char`) et chaîne de caractères (`string`).

3.1.2 Déclaration des objets et initialisation des variables

Comme on l'a vu précédemment, une donnée constante doit être déclarée en **2.**, et ce avant la déclaration des variables. Sa syntaxe est :

```
const nom_de_la_constante_1 = valeur;
      nom_de_la_constante_2 = valeur;
      ...
```

Exemple :

```
const  a=100;           {constante de type entier}
       ch='au revoir';  {constante de type chaîne de caractères}
```

Les variables sont également déclarées en **2.** (après la déclaration des constantes), et doivent être initialisées en **4.**. Pour les initialiser, il y a deux moyens possibles :

- pour les variables d'entrée : **par lecture**
  Exemple :
  `writeln('écrire les valeurs de a et b');`
  Quand le programme s'exécute, il s'affiche sur l'écran : `écrire les valeurs de a et b`. `writeln();` est une commande d'écriture.
  `readln(a,b);`
  les deux valeurs alors écrites par l'utilisateur sont affectées aux variables a et b. `readln();` est une commande de lecture.
  Remarques :
  - On aurait pu utiliser les commandes `write();` et `read();` : la seule différence est que dans ce cas, le curseur de l'écran d'affichage ne passe pas à la ligne suivante à la fin de la commande : il n'y a pas de saut de ligne.
  - **Complément sur l'instruction `write()` :**
    L'instruction `write(expression_1,...,instruction_n);` permet d'afficher une liste d'expressions (ici n expressions) sur l'écran de sortie du programme. Une expression peut être un nombre, une variable numérique, le résultat numérique entre plusieurs variables, ou une chaîne de caractères quelconque (qui peut comporter des lettres accentuées et des espaces) : dans ce dernier cas, il est nécessaire de mettre la chaîne entre deux apostrophes.
  - **Complément sur l'instruction `read()` :**
    L'instruction `read`$(x_1, \ldots, x_n)$; permet d'affecter aux variables $x_1, \ldots, x_n$ préalablement déclarées, n données numériques entrées au clavier. Ces données doivent être séparées par un caractère espace ou un retour à la ligne.
- pour les variables de sorties : **par affectation**
  Exemple :
  `x := 1;`
  Ou si a et b ont déjà été initialisées :
  `x := (a+b)/2;`

  ⚠ Si l'on écrit `a := b`, il faut que les variables a et b soient de **même type**. Ne pas confondre également le ':=' de l'affectation et le '=' réservé aux données constantes et au symbole de comparaison.

3.2 Opération entre les objets

Ces opérations dépendent du type des objets.

⚠ Les opérations sont à effectuer entre des objets de même type !

3.2.1 Opérateurs et fonctions arithmétiques

opérateurs	entrée(s)	sortie	commentaires
+ - *	réel / entier	réel / entier	opérations élémentaires
/	réel / entier	réel	le type de sortie peut donc être différent du type d'entrée
div mod	entier	entier	quotient et reste de la division euclidienne
exp ln sqrt	réel / entier	réel	`sqrt` est la racine carrée
sqr	réel / entier	réel / entier	carré
trunc	réel	entier	partie entière
abs	réel / entier	réel / entier	valeur absolue
round	réel	entier	entier le plus proche

3.2.2 Opérateurs logiques

Il y en a 3 : `not`, `or` et `and`. Les entrées comme les sorties sont de type booléen.
Les tables de vérité suivantes donnent le résultat de ces opérations ; on notera V pour vrai, F pour faux.

x	**not** x
V	F
F	V

or	V	F
V	V	V
F	V	F

and	V	F
V	V	F
F	F	F

3.2.3 Opérateurs relationnels

=, <> pour $\neq$, <= pour $\leq$, >= pour $\geq$.
Les variables d'entrées sont de type entier / réel ; la variable de sortie est de type booléen.
Ces opérations sont principalement utilisés dans les instructions conditionnelles (que l'on verra ultérieurement).

4 Instructions et instructions conditionnelles

4.1 Instructions

Une instruction peut être simple, i.e. d'une seule commande : par exemple `writeln('bonjour') ;` ou l'affectation d'une variable `s := 0 ;`.
Une instruction composée est une suite d'instructions terminées par un ' ;' : l'ensemble de ces instructions doit alors commencer par un `BEGIN` et finir par un `END ;`.

⚠ Le `BEGIN .. END .` du programme principal se termine par un '.' et non un ' ;'.

Pour des raisons de lisibilité, on utilise très souvent une indentation en décalée :

```
BEGIN
     instruction1 ;
     instruction2 ;
     instruction3
END ;
```

où `instruction1`, `instruction2` et `instruction3` peuvent être des instructions simples ou composées.
Le point virgule est en fait un séparateur d'instructions : c'est pourquoi il n'est pas nécessaire d'en insérer un à la fin de `instruction3`.

4.2 Instructions conditionnelles et `if then else`

Cette structure répond à l'attente :
"Si une relation est vraie (par exemple $x \neq 0$), alors on veut effectuer une certaine instruction (par exemple diviser par x) et sinon, on en effectue une autre."
la syntaxe est la suivante :

```
IF relation THEN
   BEGIN
        instructions B ;
   END
ELSE
   BEGIN
        instructions C ;
   END ;
instructions D ;
```

Remarques :

i. Si les instructions B ou C sont simples, le `BEGIN END` correspondant est inutile.

ii. Le `ELSE` est facultatif ; mais il ne doit pas être précédé immédiatement d'un ' ;'.

5 Boucles itératives

5.1 Boucle `for`

Cette structure est utile pour répéter une suite d'instructions n fois, lorsque n est connu à l'avance. La syntaxe est la suivante :

```
FOR i:=n TO m DO
  BEGIN
       instructions_B ;
  END ;
instructions_C ;
```

On le lit de la manière suivante : "Pour i allant de n à m, faire..." . Remarques :

i. `instructions_B` est effectué une première fois avec $i = n$, une deuxième avec $i = n + 1$, ..., puis une dernière avec $i = m$: le groupe d'instruction est exécuté $m - (n - 1)$ fois. ⚠ La variable compteur i doit être déclarée en **2.** !

ii. n et m sont deux variables de type entier, déclarées et initialisées.

iii. Si `instructions_B` est simple le `BEGIN END ;` correspondant est inutile.

iv. Si $n > m$, `instruction_B` n'est pas exécuté.

Variante :
Si l'on veut aller de m à n dans l'ordre décroissant, la syntaxe devient :

```
FOR i := m DOWNTO n DO ...
```

5.2 Boucles conditionnelles

5.2.1 Boucle `while`

Cette structure s'utilise lorsque l'on veut répéter une suite d'instructions tant qu'une certaine relation est vraie. La syntaxe est la suivante :

```
instructions_A ;
WHILE relation DO
     BEGIN
          instructions_B ;
     END ;
instructions_C ;
```

On le lit : "Tant que `relation` est vraie, faire..." . Remarques :

i. `instructions_B` peut ne pas être exécuter du tout[4].

ii. Si `instructions_B` est simple, le `BEGIN END ;` correspondant est inutile ;

iii. ⚠ Il faut s'assurer avant de lancer le programme que la relation devient fausse au bout d'un certain temps, sinon le programme ne s'arrêtera jamais !

[4] Si dès le départ la relation est fausse

5.2.2 Boucle `repeat ... until`

L'utilisation se fait lorsque l'on veut répéter une suite d'instructions jusqu'à ce qu'une relation soit vraie (c'est à dire qu'un objectif soit atteint).
La seule différence avant la boucle `while` est que dans la boucle `repeat`, `instructions_B` est exécuté avant de tester la relation au mois une fois. La syntaxe devient :

```
instructions_A ;
REPEAT
     instructions_B ;
UNTIL relation ;
instructions_C ;
```

On le lit : "Répéter ... jusqu'à `relation`". Remarques :

i. Même si `instructions_B` n'est pas une instruction simple, le `BEGIN END ;` ici est inutile car `REPEAT UNTIL` sert de délimiteur.

ii. ⚠ Il faut s'assurer avant de lancer le programme que la relation devient vraie au bout d'un certain temps, sinon le programme ne s'arrêtera jamais !

6 Fonctions et Procédures

6.1 Notion de sous-programme

La notion de sous-programme représente toute la puissance du langage `Pascal`. En fait, c'est la possibilité de structurer encore davantage le programme en créant de nouveaux ordres utilisables dans le corps du programme mais non définis dans celui-ci. Celà permet d'avoir un programme beaucoup plus lisible puisque l'on décompose ainsi le programme en actions simples ou blocs d'instructions.

Le principe :

Il peut arriver que l'on doive utiliser une même séquence d'instructions à différents endroits d'un programme. Il est alors judicieux de créer un sous-programme[5] dont le code sera défini une fois pour toutes dans l'étape **3.**, et que l'on appellera dans le corps du programme aux différents endroits souhaités.

L'avantage est que le code du programme est beaucoup plus court puisque l'on évite ainsi des répétitions de code.

Exemple : Le calcul de $C_n^k = \frac{n!}{k!(n-k)!}$ nécessite le calcul de 3 factorielles ; donc on pourra écrire une fonction factorielle en amont, et ensuite l'appeler 3 fois.

On doit pouvoir utiliser un sous-programme sans savoir ce qu'il se passe dedans : Les informations mises à notre disposition sont les paramètres d'entrées et le résultat produit.

De même, un sous-programme ne doit pas savoir ce qui se passe à l'extérieur : son seul lien avec l'extérieur sont ses paramètres d'entrées.

Un sous-programme a la même structure qu'une programme mais le `END` du corps du sous-programme est suivi d'un ';' et non d'un '.'. Il faut déclarer ces sous-programmes dans la partie **3.**, juste avant le corps du programme principal. Toutes les variables utilisées dans un sous-programme doivent être définies soit comme un paramètres d'entrée soit comme les variables locales[6].

Mais l'appel du sous-programme se fait dans le corps du programme (étape **4.**)[7].

Il y a deux sortes de sous-programmes : les fonctions et les procédures.

6.2 Les fonctions

Une fonction est un sous-programme qui fournit un résultat à partir des données qu'on lui apporte : la notion de fonction en `Pascal` est assez proche de la notion de fonction en mathématiques. Une fonction se déclare comme suit :

```
FUNCTION Nom_de_la_fonction (Nom_du_parametre1 : Type_du_parametre1) : Type_sortie ;
Déclaration des éventuelles constantes ou variables LOCALES (via CONST, VAR)
BEGIN
  Corps de la fonction (liste des instructions)
  Nom_de_la_fonction := Valeur_sortie ;
END ;
```

Remarques :

i. Une fonction ne peut avoir en sortie que des valeurs "simples" (un réel, un entier, ...)

ii. ⚠ S'il y a plusieurs paramètres d'entrées dans une fonction, ils sont séparés par un ';' mais lors de l'appel de la fonction ils seront séparés par une ','.

[5] Fonction ou Procédure

[6] Les variables du programme sont alors des variables globales

[7] Un sous-programme peut utiliser d'autres sous-programmes si ceux-ci ont été définis avant, ou peut s'utiliser soi-même : dans ce dernier cas on parle de sous-programme récursif.

6.3 Les procédures

Contrairement aux fonctions, la procédure ne fournit pas un résultat mais crée une action ou une suite d'actions (instructions) : on utilisera les procédures principalement lorsqu'on manipulera des tableaux (matrices ou autres). En effet, une fonction ne permet pas de modifier un tableau puisque son but est de renvoyer une valeur.
Sa syntaxe reste analogue à la syntaxe d'une fonction :

```
PROCEDURE Nom_de_la_procedure (Nom_du_parametre1 : Type_du_parametre1) ;
Déclaration des éventuelles constantes et variables LOCALES
BEGIN
    Corps de la procédure
END ;
```

Cette structure a une variante :

```
PROCEDURE Nom_de_la_procedure (VAR Nom_du_parametre : Type_du_parametre) ;
```

Quelle est la différence entre ces deux syntaxes ? Voyons ceci sur un exemple :
Exemple :

```
PROGRAM proc ;
var A : real ;
procedure P(x:real);
 BEGIN
   write(x);
   x := x+1;
   writeln(x);
 END;
BEGIN
  A:=5
  P(A);
  write(A);
END.
```

L'affichage est :
56
5
Une variable locale `x` est créée et elle prend la valeur 5 puis la valeur 6 et qui est ensuite détruite à la fin de la procédure. Mais la variable `A` est inchangée.
Si maintenant la première ligne de la procédure est :

```
procedure P1(VAR x : real);
```

l'affichage devient :
56
6
En effet, lors de l'exécution de la ligne de commande `P1(A)` ; la variable locale `x` n'est plus créée, c'est la variable `A` qui est utilisée pour faire les instructions. Après le traitement `A` est donc modifiée et prend la valeur 6. Cette variante au niveau syntaxe permet de changer des variables globales.
Remarques :

i. On dira dans la première syntaxe, que le paramètre d'entrée **passe par valeur**[8], alors que dans la deuxième syntaxe, il **passe par variable**[9].

ii. Dans un passage par valeur, le paramètre d'entrée peut être n'importe quoi du type défini : `P(5) ;` a un sens.
Alors que dans un passage par variable, le paramètre d'entrée doit être nécessairement une variable : ainsi dans l'exemple précédent, `P1(5)` ou `P1(x+5)` n'a aucun sens.

[8] On n'utilise que la valeur `A` dans `P`
[9] La variable `A` est utilisée et changée dans `P`

7 Les tableaux

L'utilité des tableaux :

- Pour stocker des données. Par exemple, pour stocker les résultats d'expériences aléatoires, avant de les analyse ou dans la présentation de données retraçant un historique complet de l'évolution discrète d'un paramètre.
- Pour travailler formellement sur les polynômes.
- Pour traiter des problèmes matriciels.

7.1 Déclarer un tableau

7.1.1 Tableau à une dimension

Avant de pouvoir utiliser un tableau, il faut le déclarer comme variable :

```
VAR tableau : ARRAY[deb..fin] OF (type);
```

Dans cette syntaxe, `deb` et `fin` sont de type `integer`, sous la contrainte `deb` $\leq$ `fin`. La variable `tableau` est alors un tableau composé de colonnes numérotées de `deb` à `fin`. Ce tableau comportera donc `fin - deb + 1` cases, remplies par des éléments de type `type`. Chaque case du tableau est alors une variable identifiée par `tableau`[`i`] (avec i un entier compris entre `deb` et `fin`).

Exemple : Les coordonées d'un point A du plan peuvent être représentées par le tableau suivant :

```
VAR coordA : ARRAY[1..2] OF real;
```

`coordA`[1] donne l'abscisse du point A et `coordA`[2] l'ordonnée.

Exemple : Un polynôme P de degré au plus 10 à coefficients réels peut être représenté par le tableau suivant :

```
VAR P : ARRAY[0..10] OF real;
```

Le polynôme $P(x) = 1 + 2x + 4x^3 + 5x^6 - 3x^9$ est alors représenté par le tableau :

1	2	0	4	0	0	5	0	0	-3	0

Tandis qu'un polynôme Q de degré au plus 5 à coefficients entiers (relatifs) sera déclaré par :

```
VAR P : ARRAY[0..5] OF integer ;
```

7.1.2 Tableau de dimension supérieure

Il est possible de créer des tableaux à double entrée, et même à plusieurs entrées. En pratique, on utilise le plus souvent des tableaux de dimension deux.

Exemple : Un tableau représentant une matrice `A` de $\mathcal{M}_{2,3}(\mathbb{R})$ sera déclaré par la syntaxe :

```
VAR A : ARRAY[1..2,1..3] OF real ;
```

7.2 Créer un type "tableaux"

Dans certains cas, il sera nécessaire de fabriquer un nouveau TYPE d'objet Pascal, afin de simplifier la déclaration des tableaux utilisés et surtout de pouvoir appliquer des fonctions ou des procédures à des tableaux.

Rappel 7.1. *La syntaxe d'une fonction comme d'une procédure nécessite la connaissance du type du paramètre d'entrée.*

La déclaration d'un nouveau type se fait juste après la déclaration des constantes (dans la structure général d'un programme) :

- **déclarations des types** :
  `TYPE polynome = ARRAY[0..10] OF real ;`
- Ensuite, se fait la **déclaration des variables** :
  `VAR P,Q,R : polynome;`

Dès que l'on a défini un tel type `polynome`, on peut appliquer une fonction ou une procédure à un `polynome`.

⚠ Il faut garder en mémoire qu'une fonction ne pourra jamais renvoyer un polynôme : sa sortie ne peut être qu'un type simple.
Par contre, si la procédure fait un passage par variable, elle pourra changer la variable polynôme d'entrée.

Exemple : Lorsque l'on souhaitera travailler commodément avec plusieurs matrices de $\mathcal{M}_3(\mathbb{R})$, et créer des fonctions et procédures s'y rapportant on procédera en déclarant :

```
TYPE matrice : ARRAY[1..3,1..3] OF real;
VAR A,B,C : matrice;
```

7.3 Opérations sur les tableaux

7.3.1 Opération globale

la seule opération global que Turbo-Pascal sait réaliser sur les tableaux est l'affectation en bloc des valeurs d'un tableau dans un tableau identique (mêmes dimensions et type de contenu) : si `R` et `A` sont deux tableaux, la commande : `R := A ;` remplace le contenu du tableau `R` par celui du tableau `A`.

Il n'existe aucune autre commande globale sur les tableaux !

7.3.2 Autres opérations

Compte tenu du peu d'opération globales disponibles, toutes les opérations portant sur les tableaux se font **case par case**, donc en utilisant des boucles. Ce défaut engendre souvent des temps de traitement important, en particulier pour les tableaux mult-indicés, ou de grande taille (en plus de la consommation d'espace mémoire réservé pour la création de ces tableaux).

Tableaux à 1 dimension :
Exemple :

- Déclaration :
  `VAR T : ARRAY[1..50] of REAL ;`

- Déclaration du type :
```
TYPE tab = ARRAY[1..50] of REAL ;
VAR T : tab ;
```
- Initialiser la 15-ièmme case de `T` à `0` :
```
T[15] := 0 ;
```
- Initialiser la 18-ième case de `T` à une valeur entrée par l'utilisateur :
```
WRITELN('Entrer la valeur de la case 18');
READLN(T[18]);
```
- Afficher la valeur de la case 25 du tableau :
```
WRITELN(T[25]);
```
- Initialiser tout le tableau `T` à `0` :
```
FOR k := 1 TO 50 DO T[k] := 0 ;
```
- Initialiser tout le tableau `T` à des valeurs entrées par l'utilisateur :
```
FOR k := 1 TO 50 DO
  BEGIN
    WRITELN('Entrer la valeur de la case ',k);
    READLN(T[k]);
  END ;
```
- Afficher la valeur de tout le tableau :
```
FOR k := 1 TO 50 DO WRITELN(T[k]) ;
```

Tableaux à 2 dimensions :
Exemple :

- Déclaration :
```
VAR U : ARRAY[1..50,1..40] of REAL ;
```
- Initialiser la case située à la 11-ième ligne et 23-ième colonne de `U` à `0` :
```
U[11,23] := 0 ;
```
- Initialiser la case située à la 8-ième ligne et 27-ième colonne de `U` à une valeur entrée par l'utilisateur :
```
WRITELN('Entrer la valeur de la case 8,27');
READLN(T[8,27]);
```
- Afficher la valeur de la case (25,17) du tableau :
```
WRITELN(T[25,17]);
```
- Initialiser tout le tableau `U` à `0` :
```
FOR i := 1 TO 50 DO
    FOR j := 1 TO 40 DO
        U[i,j] := 0 ;
```
- Initialiser tout le tableau `U` à des valeurs entrées par l'utilisateur :
```
FOR i := 1 TO 50 DO
    FOR j := 1 TO 40 DO
       BEGIN
         WRITELN('Entrer la valeur de la case (',i,j,')');
         READLN(U[i,j]);
       END ;
```
- Afficher la valeur de tout le tableau :
```
FOR i := 1 TO 50 DO
    FOR j := 1 TO 40 DO WRITELN(U[i,j]) ;
```

8 Simulation des lois de probabilités usuelles

Introduction :
Supposons que l'on ait une pièce truquée dont on veut déterminer la probabilité p d'obtenir face. Le moyen expérimental à notre disposition est de lancer un très grand nombre de fois la pièce, par exemple $n = 1000$, et de compter le nombre d'occurence de "faces" : n_f.
Alors le rapport $\frac{n_f}{n}$ appelé fréquence d'apparition de face, est une approximation de p, d'autant meilleure que n est grand.

Plus généralement, la fréquence de réussite d'un certain événement lors de n expériences aléatoires indépendantes successices[10], tend vers la probabilitéde cet événement quand $n \to +\infty$. Pour avoir une approximation dde cette probabilité, il faut donc effectuer un grand nombre d'expérience aléatoires (par exemple, lancé de dé ou de pièce). On peut biensûr s'armer de patience et lancer la pièce ou le dé $10\ 000$ fois, mais l'outil informatique permet de faire cette expérience en beaucoup moins de temps ! ! !
Il suffit que le langage utilisé puisse créer du hasard : pour celà, il doit posséder un générateur de nombre aléatoires.

Syntaxe :
Pour activer le générateur de nombres pseudo-aléatoires de `Turbo-Pascal`, il faut insérer dans le début du coprs du programme, la commande `Randomize ;`.
Ensuite, deux fonctions sont prédéfinies dans `Turbo-Pascal` :

i. `random ;` : sans argument retourne un réel compris entre 0 et 1.

ii. `random(n) ;` : où n est un entier > 0 retourne un entier comprius entre 0 et $n-1$ avec équiprobabilité. Autrement dit, cette fonction `random(n) ;` permet de simuler la loi uniforme sur $\{0, 1, 2, ..., n-1\}$.

[10] Répétitions d'une même expérience de départ