Informatique en ECS1

Alain GUICHET

8 octobre 2012

Table des matières

Chapitre 1

Introduction au langage de programmation Pascal

1.1 Installation de TurboPascal sur clé USB

1. Brancher la clé USB (en principe, elle détectée automatiquement). Une fenêtre s'ouvre. Choisir 'ouvrir le dossier', cocher la case 'toujours effectuer cette action' et valider.
2. Téléchargerl'archive turbo_pascal_701_fr.zip à la racine de la clé.
3. Décompresser l'archive (clic droit).
4. On peut désormais travailler avec TurboPascal version 7.0.1 : lancer l'application à l'aide de l'icône de raccourci.
5. Par défaut, les programmes seront toujours sauvés dans le dossier \pascal\mesprog.
6. En fin d'utilisation de TurboPascal, de la clé, de l'ordinateur, effectuer les actions suivantes :
   (a) i. fermer votre programme en cours (clic sur le carré vert de la fenêtre en haut à gauche)
   ii. puis fermer l'application ('file', 'exit').
   (b) i. fermer le dossier de votre clé
   ii. puis cliquer gauche sur l'icône des périphériques USB branchés, choisir la déconnexion du bon périphérique
   iii. attendre de recevoir l'autorisation de débrancher la clé puis la retirer

On ne débranche jamais une clé USB sans l'avoir indiqué à l'ordinateur

   iv. **fermer la session en cours** ou bien éteindre l'ordinateur.

1.2 Écrire un programme

Programmer consiste à écrire une suite d'ordres appelés **instructions** que l'ordinateur devra exécuter lorsqu'on le lui demandera. Pour écrire un programme (suite d'instructions), on utilise un environnement de développement intégré (ou IDE : « integrated development environment » en anglais) constitué d'un éditeur de texte (traitement de texte simplifié), d'un compilateur (traducteur en langage machine) et d'un gestionnaire d'exécution (pour vérifier le fonctionnement du programme). Nous utiliserons donc l'IDE nommé TurboPascal.

Lancez le programme TurboPascal. L'IDE s'ouvre directement sur l'éditeur ; on peut maintenant écrire le programme souhaité. Le principe de la création d'un programme est le suivant :

- demander à l'utilisateur de saisir des données,
- traiter les données (i.e. effectuer des calculs sur ces données),
- afficher les résultats du traitement.

Il s'agit, pour cela, d'utiliser un langage simultanément compréhensible par un humain (le programmeur) et par la machine. Le langage utilisé cette année est le langage Pascal (langage créé initialement pour l'apprentissage de la programmation) qui est très proche de celui de vos calculatrices et dont les instructions essentielles sont en anglais.

Ouvrez un nouveau document (Fichier/Nouveau) puis recopier exactement le programme suivant :

```
PROGRAM mon premier programme ;
VAR m,n : integer ;
BEGIN
write('Donner un premier nombre entier : ') ; readln(m)
write('Donner un second nombre entier  : ') ; readln(n) ;
writeln('Somme : ',m+n,'   Différence : ',m-n) ;
```

```
writeln('Produit : ',m*n,'   Quotient : ',m/n) ;
writeln('Quotient entier : ',m DIV n,'   Reste : ',m MOD n) ;
END ;
```

Vous pouvez constater que certains mots s'écrivaient en blancs (tous ceux écrits en majuscules dans le programme) : c'est bon signe, la machine a reconnu en eux un mot important du programme appelé **mot réservé** [1]. Dans le cas où ces mots réservés ne s'écrivent pas en blanc, il est probable que vous avez commis une faute d'orthographe ou oublié d'insérer un espace pour séparer des mots.

1.3 La sauvegarde

Avant toute nouvelle manœuvre, il est *fondamental* de **sauvegarder** votre programme (nul n'est à l'abri d'une coupure de courant). Nous sauvegarderons toujours nos documents sur une clé USB dans le répertoire créé à cet effet :

- cliquez sur File / Save (ou bien appuyer sur la touche F2),
- donnez un nom *sans accent, ni espace, ni point* relativement explicite, comme par exemple `CH1_1.PAS` (pour chapitre 1, paragraphe 1 ; l'extension `.PAS` est facultative [2]).

1.4 La compilation

Le programme étant écrit, on cherche à le faire exécuter par la machine. Pour cela, l'ordinateur doit tout d'abord le **compiler**, c'est à dire le traduire dans un langage totalement compréhensible par le processeur ; on clique [3] successivement sur :

Compile / Compile

Vous constatez donc qu'un bandeau rouge contenant un message d'erreur apparaît en haut de la fenêtre, lisez et traduisez le texte du message (en anglais...). On corrige cette erreur en remplaçant :

```
PROGRAM mon premier programme ;
```

par :

```
PROGRAM mon_premier_programme ;
```

ou bien par :

```
PROGRAM MonPremierProgramme ;
```

puisque *le nom du programme doit être d'un seul tenant.*

On relance la compilation et là encore elle provoque une erreur (« `; expected` ») car il manque un « ; » à la fin de la ligne qui précède le numéro de ligne indiqué.

Une troisième compilation s'arrête car la machine n'a pas trouvé la fin du programme (elle cherche autre chose à compiler mais ne trouve plus rien). La correction à apporter consiste à remplacer :

```
END ;
```

par :

```
END.
```

le « . » marquant la fin du programme.

Une dernière compilation s'effectue sans erreur. Vous remarquez donc que toutes les erreurs doivent être corrigées et que la machine est très pointilleuse sur l'orthographe.

1.5 L'exécution

On peut ensuite l'**exécuter** : on clique sur « Run » [4] [5]. Vous êtes désormais sur l'écran de « travail » du programme. Saisissez les valeurs 40000 et 20000 et constatez que vous revenez à l'écran d'édition. En fait, il s'est passé quelque chose mais cela s'est déroulé trop vite pour votre œil. Pour visualiser les résultats, il faut appuyer sur « Alt+F5 » [6]. L'idéal pour éviter que ce désagrément ne se reproduise est d'insérer systématiquement en avant dernière ligne de chaque programme l'instruction :

1. Vous pouvez vous dispenser d'écrire en majuscule les mots réservés, c'est une simple question de lisibilité de ce document pour le programmeur.
2. Si l'extension du fichier n'est pas `.PAS` (c'est à dire s'il y a un point dans le nom du fichier) alors les mots réservés et les messages ne seront plus dans leurs couleurs distinctives.
3. La compilation est aussi lancée par la combinaison de touches « Alt+F9 ».
4. L'exécution est aussi lancée par la combinaison de touches « Ctrl+F9 ».
5. On peut cliquer sur « Run » sans avoir préalablement compilé le programme, l'ordinateur se chargeant automatiquement de lancer cette tâche.
6. C'est une touche de bascule entre l'éditeur de texte et l'écran de visualisation d'exécution.

```
readln ;
```

puis de recompiler de réexécuter.

Vous visualisez désormais des résultats qui sont tout particulièrement farfelus. Ce n'est ni tout à fait de votre faute ni tout à fait celle de l'ordinateur. Ces bizarreries seront expliquées au chapitre 2. Vous pouvez lancer une nouvelle exécution avec les valeurs 100 et 23, tout rentrera dans l'ordre.

Remarque fondamentale : lorsque vous écrivez le programme dans l'éditeur de texte, vous jouez le rôle de la machine ; lorsque vous testez votre programme, vous êtes un simple utilisateur qui ne sait pas nécessairement comment le programme fonctionne. Les instructions `readln` et `writeln` (lire et écrire) s'interprètent donc au sens de la machine et non au sens de l'utilisateur (c'est la machine qui lit et écrit).

1.6 Quelques commentaires

Le programme définitif est donc le suivant :

```
PROGRAM mon_premier_programme ;
VAR m,n : integer ;
BEGIN
write('Donner un premier nombre : ') ; readln(m) ;
write('Donner un second nombre  : ') ; readln(n) ;
writeln('Somme : ',m+n,'   Différence : ',m-n) ;
writeln('Produit : ',m*n,'   Quotient : ',m/n) ;
writeln('Quotient entier : ',m DIV n,'   Reste : ',m MOD n) ;
readln ;
END.
```

- La première ligne consiste à donner un nom au programme (sans espace, sans accent, sans symbole diacritique). Elle est constituée d'une instruction et se termine par un « ; » comme toute instruction. Cette ligne est facultative.
- La deuxième ligne est la ligne de déclaration des **variables globales** (celles qui peuvent être utilisées partout entre leur déclaration et la dernière ligne) dont le programme aura besoin. Cette ligne est fondamentale. Elle donne des noms (i.e. des références) à des zones de stockage de données numériques ou non numériques dans la mémoire vive de l'ordinateur et précise la nature de l'objet stocké, ce que l'on appelle le **type** en informatique. Ici, la ligne se traduit en langage mathématique par la phrase « Soit m et n deux entiers relatifs ». Remarquez la présence du « ; » qui marque la fin de l'instruction.
- La ligne 3 marque le début du programme. Ce n'est pas une instruction (mais plutôt une balise), donc pas de « ; » en fin de ligne.
- La ligne 4 est composée de deux instructions (chacune se terminant par un « ; »). La première donne l'ordre (`write`) à l'ordinateur d'écrire à l'écran le message se situant entre les apostrophes [7] *et de laisser le curseur à la suite de ce qu'il a affiché*. La seconde donne l'ordre (`readln`) à l'ordinateur de lire le nombre écrit par l'utilisateur au clavier, de placer ce nombre dans la case mémoire référencée « `m` » *et de placer le curseur en début de ligne suivante* [8]. Attention, aucune vérification d'orthographe ou de grammaire n'est effectuée par le compilateur dans les messages que l'ordinateur doit afficher.
- La ligne 5 est du même acabit que la 4.
- La ligne 6 est formée d'une unique instruction (`writeln`) donnant l'ordre d'afficher quatre informations à l'écran les unes à la suite des autres (*sans les séparer par des espaces*) et de placer le curseur en début de ligne suivante [9]. Les quatre informations (séparées par des « , ») sont constituées d'un message, du résultat d'une opération sur des **valeurs de variables** (le contenu de la zone référencée), d'un second message et d'un autre résultat d'opération.
- Les lignes 7 et 8 sont du même acabit que la 6. Remarquez toutefois la présence d'opérations particulières (`DIV` et `MOD`) dont le commentaire qui précède explique la nature (par exemple $12 = 8 \times 1 + 4$ donc $12\,\texttt{DIV}\,8 = 1$ et $12\,\texttt{MOD}\,8 = 4$).
- La ligne 9 permet à l'utilisateur du programme de lire tranquillement ses résultats. Lorsqu'il a terminé, il doit appuyer sur « Enter » pour valider la fin du travail.
- La ligne 10 indique au compilateur la fin du programme (c'est encore une balise).

Pour le moment, la syntaxe générale d'un programme en langage Pascal est :

```
PROGRAM nom_de_programme ; {aucun espace ni symbole diacritique}
VAR liste_des_variables_de_type_entier : integer ; {, comme séparateur}
    liste_des_variables_de_type_réel : real ; {, comme séparateur}
    autres_variables : autres_types ;
BEGIN {début du programme, là où commence l'exécution}
{saisie des données}
```

7. Notez que les messages qui s'afficheront à l'écran lors de l'exécution sont affichés dans une couleur différente de celle des instructions afin de bien les distinguer.

8. « `ln` » signifie line (english word, of course).

9. Remarquez la légère différence entre les instructions `write` et `writeln`.

```
   instructions_des_saisies ; {au moins un readln()}
{traitement des données}
   instructions_de_traitement ; {des calculs en règles générale}
{affichage des résultats, éventuellement au cours du traitement}
   instructions_des_affichages {au moins un writeln()}
readln ; {pour prendre le temps de lire les résultats}
END. {un point pour terminer la compilation}
```

Une dernière remarque : en Pascal, on place entre accolades tous les commentaires jugés utiles par le programmeur pour qu'il puisse faire comprendre ses choix de programmation et qu'il puisse se remettre rapidement dans son programme après l'avoir laissé de côté quelques temps.

1.7 Exercices

Exercice 1.1 (Année de naissance)
Écrire un programme qui :

1. affiche le message « `Donnez-moi votre âge :` »,
2. attend que l'utilisateur donne son âge,
3. affiche le message « `Précisez-moi l'année actuelle :` » [10],
4. attend que l'utilisateur donne l'année en cours,
5. affiche le message « `Votre année de votre naissance est :` » suivi du résultat du calcul donnant l'année de naissance de l'utilisateur.

Exercice 1.2 (Calculs de quotients et restes)
Écrire un programme qui saisit trois nombres entiers (`a1`, `a2` et `a3`) puis calcule et affiche le reste entier de la division du quotient entier de `a1` par `a2` avec le quotient entier de `a2` par `a3`.
Tester le programme avec `a1=1000`, `a2=238` et `a3=44` (le résultat à obtenir est `4`).

10. Attention, pour afficher l'apostrophe de « `l'année` », il faut écrire « `l''année` » (avec deux apostrophes) dans le programme.

Chapitre 2

Les opérations sur les types

2.1 Notion de type

Comme nous l'avons vu dans le chapitre 1, toute zone de stockage temporaire utile à un programme doit être déclarée en début de programme. Une déclaration comporte deux renseignements : le nom de la zone (appelée **variable globale**) et la nature du contenu de la zone (appelée **type**) afin de réserver un emplacement plus ou moins grand dans la mémoire de l'ordinateur. Quatre types sont à connaître pour le programme d'informatique de ECS [1].

- Le type « nombre entier », noté `integer`, correspond aux entiers relatifs avec une restriction importante : les entiers sont dans l'intervalle $[\![-32768, 32767]\!]$ (l'ordinateur ne sait pas gérer l'infini).
  De plus, l'ensemble est cyclique au sens de : « l'entier qui suit 32767 est l'entier -32768 ». La plupart des résultats farfelus que vous pouvez obtenir à l'exécution d'un programme sont dûs à cette restriction (comme lors la première exécution du programme au chapitre 1).
- Le type « nombre réel », noté `real`, correspond en fait à des nombres décimaux (l'ordinateur ne sait pas gérer une infinité de décimales) dans une plage relativement étendue.
  Ces nombres sont toujours affichés, par défaut, sous la forme scientifique (0,00123456789 est écrit `1.23456789E-03`).
- Le type « tableau de nombres », noté `ARRAY`, à une ou plusieurs lignes que l'on verra aux chapitres 3 et 8.
- Le type booléen, noté `boolean`, que l'on verra au chapitre 4.

2.2 Affectation

Le contenu d'une variable est appelé **valeur de la variable**. Pour modifier cette valeur, on a déjà vu au chapitre 1 qu'il suffit de saisir sa valeur au clavier à l'aide de la procédure `readln`, l'inconvénient étant que seul l'utilisateur peut modifier cette valeur avec cette technique. Toutefois, un programme peut quand même modifier la valeur d'une variable en lui **affectant** une nouvelle valeur.

Définition 2.1 :
L'**affectation** consiste à placer une valeur dans la variable soit directement par le programme soit comme résultat d'une combinaison d'opérations effectuées par le programme :

- soit de manière **absolue** en donnant la valeur exacte, avec des calculs ou non (la variable `a` étant de type `integer`) :

  `a := 1 ;` ou `a := (100 - 3) DIV 4 + 1 ;`

  (avec les règles de priorités en usage en mathématiques), le symbole « `:=` » devant s'interpréter comme suit : mettre le résultat du calcul dans la zone mémoire référencée `a`.
- soit de manière **relative**, en réutilisant la valeur précédente (la variable `x` étant de type `real`) :

  `x := x + 0.5 ;`

  consiste à augmenter la valeur de `x` de 0,5. On interprète cela comme suit : prendre la valeur de la variable `x` (le `x` du membre de droite), lui ajouter 0,5 et mettre le résultat dans la variable référencée `x` (le `x` du membre de gauche).

Remarque (Attention lors de la déclaration du type d'une variable) :
La ligne de programme précédente crée une erreur de compilation (« `type mismatch` » [2]) si la variable `x` est de type `integer`. En effet, l'opération `x := x+0.5` génère nécessairement un résultat de type `real` (puisque 0,5 ne peut pas être entier) qui ne peut se « loger » dans une variable de type entier. Par contre, tout entier est implicitement considéré comme réel si nécessaire.

1. D'autres types existent en langage Pascal et on peut même en créer, mais ce ne sera pas notre propos ici.
2. C'est l'erreur de programmation la plus souvent rencontrée avec l'oubli du « `;` » en fin d'instruction.

Exemple :
Un utilisateur fournit les nombres 2 et 3 au programme qui suit. Qu'affiche la machine ?

```
PROGRAM exemple_affectations ;
VAR a,b : real ;
BEGIN
write('Donner a : ') ; readln(a) ;
write('Donner b : ') ; readln(b) ;
a := 2*a+b-1 ;
b := a-3*b+1 ;
writeln('a = ',a,' et b = ',b) ;
readln ;
END.
```

2.3 Opérations et fonctions usuelles

Définition 2.2 :
- Les opérations usuelles *entre deux nombres entiers* (c'est à dire que le résultat est de type `integer`) sont :

  $$+ \quad - \quad * \quad \texttt{DIV} \quad \texttt{MOD}$$

  où `DIV` désigne le quotient (entier) de la division euclidienne et `MOD` le reste (entier) de la division euclidienne. Le résultat de chacune de ces opérations est donc de type `integer` pourvu que les opérandes le soient (sinon il y a une erreur de compilation). Les règles de priorités usuelles des opérations s'appliquent (`*`, `DIV` et `MOD` sont prioritaires sur `+` et `-`).
- Les opérations usuelles *entre deux nombres réels* (ou bien entre un nombre réel et un nombre entier, ou bien encore entre un nombre entier et un nombre réel, ou bien même entre deux nombres entiers) sont :

  $$+ \quad - \quad * \quad /$$

  avec les règles usuelles de priorités des opérations. Remarquez que la barre de division génère toujours un résultat de type `real` même si les deux opérandes sont de type `integer`.
- Les fonctions usuelles sur les réels (ou les entiers) sont :
  - `sqrt()` (racine carrée),
  - `abs()` (valeur absolue),
  - `ln()` et `exp()` (devinez !),
  - `sin()` et `cos()`.

  Les fonctions puissances n'existent pas, la fonction tangente non plus.
- On peut convertir un `real` en un `integer` à l'aide de la fonction partie entière `trunc()` ou bien arrondi à l'entier le plus proche `round()` (cette dernière fonction n'est pas explicitement au programme mais peut se révéler utile).

Remarques :
- Attention à ne pas confondre les opérations de divisions `DIV` et `/`. Ainsi :
  - `5 DIV 3` donne le résultat `1` (puisque 5=3*1+2 et 2<3).
  - `5/3` donne le résultat `1.666666667E00`.
  - `5.1 DIV 3.2` provoque une erreur de compilation (que les valeurs soient écrites comme cela ou bien que les valeurs soient contenues dans des variables de type `real`).
  - `5.1/3.2` donne le résultat `1.593750000E00`.
- Attention aussi à la gestion des types en sortie de calculs. Ainsi :
  - `sqrt(4)` donne le résultat `2.000000000E00` (réel) et non `2` (entier).
  - `exp(ln(2))` donne le résultat `2.000000000E00` (réel) et non `2` (entier).

2.4 Exercices

Exercice 2.1 (permutation des valeurs de deux variables)
Écrire un programme qui, d'abord saisit deux entiers (stockés dans des variables notées `a` et `b`), ensuite **permute** les valeurs de ces deux variables (c'est à dire que la variable `a` **doit contenir** au final la valeur initiale de `b` et vice versa) et enfin affiche ces nouvelles valeurs. On rappelle que *toute modification de la valeur d'une variable entraîne la perte irrémédiable de l'ancienne valeur.*
Un exemple d'exécution du programme est :

```
Donner un entier a : 3
Donner un entier b : -5
On permute les valeurs
```

```
Nouvelle valeur de a : -5
Nouvelle valeur de b : 3
```

Exercice 2.2 (Différentes notions de moyennes)
Écrire un programme qui, successivement :

1. saisit 3 nombres entiers positifs (dans des variables nommées `n1`, `n2` et `n3`),
2. calcule et stocke la moyenne arithmétique [3] (dans la variable `a`), la moyenne géométrique [4] (dans la variable `g`), la moyenne harmonique [5] (dans la variable `h`) et la moyenne quadratique [6] (dans la variable `q`),
3. affiche ces quatre nombres sur quatre lignes successives (avec un message indicatif explicite).

Tester le programme avec `n1=1`, `n2=2` et `n3=3`. On doit obtenir les résultats suivants :

```
a=2.000000000E00
g=1.817120593E00
h=1.636363636E00
q=2.160246899E00
```

Exercice 2.3 (Quelques fonctions)
Écrire un programme qui saisit un réel x et affiche, successivement, son image par les fonctions [7] [8] :

$$x \mapsto \ln\left(\frac{\sin(x)}{x}\right)$$

$$x \mapsto x\,e^x$$

$$x \mapsto x^3 + 6x^2 - 9x + \pi$$

$$x \mapsto \frac{x\sqrt{x+2}}{(x+1)(x-3)}$$

$$x \mapsto \sqrt{\lfloor x^2 \rfloor}$$

3. La moyenne usuelle.
4. La racine n-ième du produit des n nombres dont on calcule la moyenne, c'est à dire l'exponentielle de la moyenne arithmétique des logarithmes des n nombres.
5. L'inverse de la moyenne arithmétique des inverses.
6. La racine carrée de la moyenne arithmétique des carrés.
7. Le symbole π n'existe pas, mais il existe une constante (variable que vous ne pouvez pas modifier) prédéfinie nommée `pi` et donc la valeur est le nombre attendu.
8. Le symbole $\lfloor \ldots \rfloor$ désigne la fonction partie entière.

Chapitre 3

Les tableaux et l'instruction itérative FOR...TO...DO...

3.1 Exemple : calcul d'un terme d'une suite récurrente

L'objectif est de calculer (de manière approchée bien sûr) le terme de rang n de la suite définie par :

$$u_0 \in \mathbb{R} \qquad \text{et} \qquad \forall k \in \mathbb{N},\ u_{k+1} = \sqrt{1+u_k^2} - 1$$

Recopiez et testez le programme qui suit.

```
PROGRAM suite_recurrente_tableau ;
VAR u : ARRAY [0..1000] OF real ;
    k,n : integer ;
BEGIN
write('Donner u0 : ') ; readln(u[0]) ;
write('Donner n  : ') ; readln(n) ;
FOR k := 0 TO n-1 DO u[k+1] := sqrt(1+u[k]*u[k])-1 ;
FOR k := 0 TO n DO writeln('u',k,' = ',u[k]) ;
readln ;
END.
```

Remarques :

- L'entier n fourni par l'utilisateur doit être un élément de $[\![0, 1000]\!]$.
- On commence par calculer tous les termes de la suite du rang 1 au rang n : on dit que l'on effectue une **boucle** ou bien que l'on **itère** l'instruction de calcul par récurrence.
  Ensuite on affiche, un par un, tous les termes du rang 0 au rang n (ici encore on dit que l'on itère l'instruction d'affichage ou bien que l'on effectue une boucle pour l'affichage).
- Toutefois, il est possible d'afficher les termes au fur et à mesure des calculs ; pour cela on remplace les lignes 7 et 8 du programme précédent par les lignes qui suivent :

```
writeln('u0 = ',u[0]) ;
FOR k := 0 TO n-1 DO BEGIN
                     u[k+1] := sqrt(1+u[k]*u[k])-1 ;
                     writeln('u',k+1,' = ',u[k+1]) ;
                     END ;
```

- Dans la mesure où le programme doit effectuer deux instructions à chaque itération (calcul et affichage [1]), il faut le préciser à la machine par un marqueur de début (`BEGIN`) et un marqueur de fin (`END ;`). On dit que l'on a ainsi une **instruction composée** ou un bloc d'instructions.
- On peut aussi remplacer la ligne :

```
FOR k := 0 TO n-1 DO u[k+1] := sqrt(1+u[k]*u[k])-1 ;
```

  par la ligne :

```
FOR k := 1 TO n DO u[k] := sqrt(1+u[k-1]*u[k-1])-1 ;
```

1. Il est impossible de faire effectuer ces deux instructions en une seule. Tout travail à faire effectuer à un programme doit être bien décomposé en travaux « élémentaires ».

3.2 Syntaxe de la déclaration d'un tableau

Définition 3.1 :
Un **tableau** se déclare comme une variable, puisque c'est une liste de variables numérotées. La déclaration s'effectue de la manière suivante :

```
nom_tableau : ARRAY [debut..fin] OF type_voulu ;
```

Remarques :

- `debut` et `fin` sont deux nombres de type `integer` avec `debut<fin` ;
- `debut` et `fin` ne peuvent pas être des variables (mêmes entières) car un tableau doit avoir une taille fixe et définie à l'avance (l'ordinateur ne peut pas réserver un nombre inconnu d'emplacements de mémoire) ;
- les deux points (`..`) sont obligatoires ;
- chaque élément du tableau est une variable du type `type_voulu` (l'un des types usuels : `integer`, `real`, `boolean`) ;
- l'élément numéro `k` ($k \in [\![\texttt{debut}, \texttt{fin}]\!]$) du tableau `nom_tableau` est la variable `nom_tableau[k]` qui s'utilise comme toutes les autres variables pour la saisie, l'affectation et l'affichage ;
- les instructions `readln(nom_tableau)` et `writeln(nom_tableau)` produisent une erreur de compilation car il faut saisir et afficher un tableau élément par élément.

3.3 Syntaxe de l'instruction itérative `FOR...TO...DO...`

Définition 3.2 (instruction unique à itérer) :
Lorsque l'on a qu'une seule instruction à itérer (i.e. répéter) comme c'est le cas en début d'exemple précédant), la syntaxe est la suivante :

```
FOR variable_muette := valeur_debut TO valeur_fin DO instruction ;
```

Remarques :

- La variable `variable_muette` est presque toujours de type `integer` (elle peut, exceptionnellement, être de type `boolean`). Cette variable est appelée **compteur** de boucle (devinez pourquoi !).
- Les expressions `valeur_debut` et `valeur_fin` sont obligatoirement des nombres ou des variables de type `integer`.
- Le principe de la boucle est **d'incrémenter** le compteur `variable_muette`, c'est à dire que `instruction` est réalisée successivement :
  - pour `variable_muette=valeur_debut`,
  - puis pour `variable_muette=valeur_debut+1`,
  - et ainsi de suite jusqu'à ce que `variable_muette=valeur_fin`.
- Lorsque la dernière itération de `instruction` est achevée, la machine passe à l'instruction qui suit.
- Lorsque `valeur_debut>valeur_fin`, l'instruction `instruction` n'est jamais réalisée et la machine passe directement à l'instruction suivante (c'est le cas dans l'exemple qui précède si on fournit la valeur 0 au programme).
- La tradition veut que `nom_variable` soit `i` ou `j` ou `k` (comme pour les notations de sommes ou de produits). Il convient donc de réserver ces noms de variables pour effectuer des itérations.
- Il est possible de **décrémenter** le compteur (on compte à rebours). Pour cela, on remplace le mot réservé `TO` par `DOWNTO`.
- Enfin, n'oubliez jamais de sauvegarder votre programme avant de l'exécuter : si la boucle est mal programmée, elle peut alors ne jamais se terminer et l'ordinateur paraît ainsi « planté » [2].

Définition 3.3 (instructions multiples ou instruction composée à itérer) :
Lorsque l'on souhaite grouper plusieurs instructions (donc au moins deux), on délimite ce groupe d'instructions, appelé **instruction composée**, par « `BEGIN ... END ;` » comme suit :

```
BEGIN
instruction_1 ;
instruction_2 ;
...
instruction_fin ;
END ;
```

La syntaxe de l'itération de ce type de bloc est donc :

```
FOR nom_variable := valeur_debut TO valeur_fin DO
    BEGIN
    instruction_1 ;
    ...
    instruction_fin ;
    END ;
```

2. En appuyant simultanément sur « Ctrl+Pause », on arrête la boucle infernale et on retourne sous l'éditeur de texte à l'endroit même où elle s'est arrêtée. Si on relance l'exécution, on reprend la boucle depuis son point d'arrêt.

Remarques :

- Les commentaires qui précédent s'appliquent aussi dans ce cas.
- Marquez l'indentation (décalage puis alignement des instructions correspondant à un même niveau d'exécution) pour faciliter la (re)lecture et la correction d'éventuelles erreurs.
- Si vous oubliez la délimitation de bloc (`BEGIN/END`), seule `instruction_1` sera itérée. Les autres instructions ne seront exécutées qu'à l'issue de toutes les itérations de `instruction_1` : elles ne seront donc exécutées qu'une seule fois, puisqu'elles ne font pas partie de la boucle.

3.4 Exercices

Exercice 3.1 (autres suites récurrentes)

Pour chacune des suites, écrire un programme qui saisit son premier terme u_0 et un rang n puis calcule et affiche le terme u_n :

1. $\forall n \in \mathbb{N},\ u_{n+1} = u_n e^{-u_n}$
2. $\forall n \in \mathbb{N},\ u_{n+1} = \sqrt{(n+1)u_n}$
3. $\forall n \in \mathbb{N},\ u_{n+1} = \dfrac{3u_n + u_n^2}{2^n}$ (*indication* : interdiction d'utiliser l'exponentielle mais utiliser une suite auxiliaire pour le calcul de 2^n)

Exercice 3.2 (suite récurrente linéaire d'ordre 2)

Écrire un programme qui saisit les deux premiers termes u_0 et u_1 et un entier $n \geqslant 2$ puis calcule et affiche le terme u_n de la suite définie par :

$$u_0 = 0 \qquad u_1 = 1 \qquad \text{et} \qquad \forall n \in \mathbb{N},\ u_{n+2} = u_{n+1} + u_n$$

Calculer u_5 puis u_{100}.

Exercice 3.3 (calculs de moyennes)

Reprendre l'exercice de calculs de moyennes (exercice 2.2). Modifier le programme pour :

1. saisir n nombres stockés dans le tableau appelé `note` ;
2. calculer la moyenne arithmétique des n éléments de `note` ;
3. calculer les autres moyennes.

Indication : en posant $S_k = x_1 + \cdots + x_k$ pour tout entier k, il convient de remarquer que $S_1 = x_1$ et $S_{k+1} = S_k + x_{k+1}$ pour tout entier k.

Exercice 3.4 (permutation des éléments d'un tableau)

Écrire un programme qui :

1. saisit un entier n puis un tableau (noté x) de n réels,
2. permute les éléments de ce tableau de sorte que chaque élément `x[i]` (pour $i \in [\![1, n-1]\!]$) voit son contenu remplacé par la valeur de l'élément `x[i+1]` et l'élément `x[n]` par la valeur de `x[1]`,
3. affiche les éléments du tableau ainsi actualisé.

Exercice 3.5 (triangle de Pascal)

Écrire un programme qui détermine puis affiche les lignes 0 à n du triangle de Pascal, l'entier n étant fourni par l'utilisateur. On rappelle que :

$$\forall k \in [\![0, n]\!],\ \binom{k}{0} = \binom{k}{k} = 1,$$

$$\forall k \in [\![1, n]\!],\ \forall i \in [\![1, k-1]\!],\ \binom{k}{i} = \binom{k-1}{i} + \binom{k-1}{i-1}$$

Pour cela, on utilisera un tableau à double entrée du type :

```
pascal : ARRAY [0..10, 0..10] OF longint;
```

utilisé avec les variables `pascal[k,i]` (pour des entiers `k` et `i` convenables) contenant l'entier $\binom{k}{i}$.

Exercice 3.6 (tirage au sort)

L'ordinateur est muni d'un générateur de nombres aléatoires. Pour l'utiliser, une instruction et une fonction sont à connaître :

```
randomize ;
```

qui initialise le générateur (imaginez les boules de loto que l'on brasse) et :

`random(n)` ;

fonction qui donne un entier choisi au hasard dans $[\![0, n-1]\!]$.
On joue à *pile ou face* n fois consécutivement à l'aide d'une pièce non truquée. On suppose que, pour l'ordinateur, *pile* est représenté par 1 et *face* par 0.
Écrire un programme qui affiche les résultats des différents lancers puis le nombre de *pile* obtenu au cours des n lancers (l'entier n étant fourni par l'utilisateur, les résultats des lancers étant stockés dans un tableau que l'on nommera `lancer`).

Chapitre 4

Les booléens et l'instruction conditionnelle IF...THEN...ELSE...

4.1 Premier exemple

```
PROGRAM booleens ;
VAR reponse : boolean ;
    a,b : integer ;
BEGIN
write('Donner un entier a : ') ; readln(a) ;
write('Donner un entier b : ') ; readln(b) ;
reponse := (a=b) ; writeln('a=b ? ',reponse) ;
reponse := (a<b) ; writeln('a<b ? ',reponse) ;
reponse := (a<>0) ; writeln('a non nul ? ',reponse) ;
reponse := odd(a) ; writeln('a impair ? ',reponse) ;
reponse := NOT odd(b) ; writeln('b pair ? ',reponse) ;
reponse := (odd(a) AND odd(b)) ; writeln('a*b impair ? ',reponse) ;
reponse := ((a<=1) OR (a>=5) OR (b<=0) OR (b>=2)) ;
writeln('Le couple (a,b) n''est pas dans ]1,5[x]0,2[ ? ',reponse) ;
readln ;
END.
```

4.2 Le type `boolean`

Définition 4.1 :
Le quatrième et dernier type à connaître est appelé `boolean` (du nom du logicien anglais Georges BOOLE) qui est un ensemble à deux éléments : `FALSE` et `TRUE`. On utilise, souvent de manière implicite, ce type `boolean` pour évaluer une proposition, c'est à dire « la proposition est-elle vraie ou fausse ? », le résultat étant toujours de type booléen. Les propositions sont écrites à l'aide des opérateurs :

- de comparaison :
  - égalité et non égalité : `=` et `<>` (en maths : $\neq$)
  - inégalités strictes : `<` et `>`
  - inégalités larges : `<=` (en maths : $\leqslant$) et `>=` (en maths : $\geqslant$)
- de conjonction des propositions :
  - négation : `NOT`
  - et, ou (inclusif) : `AND`, `OR`

Théorème 4.1 (Ordre de priorité des opérateurs sur les booléens) :
L'ordre de priorité de ces différents opérateurs est (du plus prioritaire au moins prioritaire) :

1. `NOT`
2. `AND`, `*`, `/`, `DIV`, `MOD`
3. `OR`, `+`, `-`
4. `=`, `<>`, `<`, `>`, `<=`, `>=`

Remarques :

- Le compilateur ne comprend qu'une relation à la fois : la double inégalité $a \leqslant x \leqslant b$ se traduira non pas par `a<=x<=b` mais par `(a<=x) AND (x<=b)`, puisque la proposition `a=b AND b=c` est interprété par le compilateur

comme `a=(b AND b)=c` et provoque ainsi une erreur de compilation. Par conséquent, il ne faut pas hésiter à abuser des parenthèses afin de bien se faire comprendre du compilateur.
– La procédure de lecture au clavier (`readln`) ne peut utiliser un argument de type booléen.

Exemple :

À quoi sert le programme qui suit ?

```
PROGRAM equation_ax_plus_b_egal_0 ;
VAR a,b : real ;
    all,one,zero : boolean ;
BEGIN
write('Donner la valeur de a : ') ; readln(a) ;
write('Donner la valeur de b : ') ; readln(b) ;
all := (a=0) AND (b=0) ;
one := (a<>0) ;
zero := (a=0) AND (b<>0) ;
writeln('ALL : ',all) ; writeln('ONE : ',one) ; writeln('ZERO : ',zero) ;
readln ;
END.
```

4.3 Deuxième exemple

Tester le programme suivant :

```
PROGRAM minimum ;
VAR a,b,c,min : real ;
BEGIN
write('Donner un réel a : ') ; readln(a) ;
write('Donner un réel b : ') ; readln(b) ;
write('Donner un réel c : ') ; readln(c) ;
min := a ;
IF b<min THEN min := b ;
IF c<min THEN min := c ;
writeln('Le minimum de ces trois réels est : ',min) ;
readln ;
END.
```

Exemple :

1. Modifiez ce programme pour qu'il calcule *aussi* le maximum des trois réels saisis.
2. Modifiez encore ce programme pour qu'il affiche le rang du maximum et celui du minimum.

4.4 Troisième exemple

Tester le programme suivant :

```
PROGRAM equation_degre_1 ;
VAR a,b,x : real ;
BEGIN
writeln('Résolution de l''équation ax+b=0') ;
write('Donner le coefficient a : ') ; readln(a) ;
write('Donner le coefficient b : ') ; readln(b) ;
IF a<>0
   THEN BEGIN
        x := -b/a ;
        writeln('Une solution unique : ',x) ;
        END
   ELSE IF b=0 THEN writeln('Tous les réels sont solution')
               ELSE writeln('Pas de solution') ;
readln ;
END.
```

4.5 Syntaxe de l'instruction conditionnelle `IF...THEN...ELSE...`

Définition 4.2 :
Le langage Pascal est doté d'une **instruction conditionnelle** qui permet de réaliser une ou plusieurs instructions lorsqu'une certaine condition est vraie mais qui peut aussi réaliser, au besoin, d'autre(s) instruction(s) lorsque cette même condition est fausse :

```
IF condition THEN instruction_si_vrai [ELSE instruction_si_faux] ;
```

ou encore :

```
IF condition
   THEN BEGIN
        instruction1_si_vrai ;
        ...
        END
  [ELSE BEGIN
        instruction1_si_faux ;
        ...
        END] ;
```

- `condition` est soit une variable booléenne soit une proposition dont le programme évaluera la véracité.
- `instruction_si_vrai` et `instruction_si_faux` désignent une instruction unique. Si plusieurs instructions sont à réaliser dans l'un ou l'autre des cas, il faut utiliser les délimiteurs `BEGIN...END` ; comme dans la seconde version ci-dessus.
- `instruction_si_vrai` est exécutée si et seulement si la proposition `condition` est vraie, `instruction_si_faux` est exécutée dans le cas contraire : seule l'une des 2 instructions est réalisée avant de passer à la suite.
- La partie notée entre crochets `ELSE instruction_si_faux` est facultative (les crochets ne doivent pas être insérés dans le code réel).
- **On ne met jamais de ; avant `ELSE`** (provoque une erreur de compilation).
- Pour faciliter la lecture du code et la recherche d'erreurs éventuelles, il est plus que souhaitable de respecter les alignements comme ci-dessus.

4.6 Exercices

Exercice 4.1 (Pile ou Face)
Écrire un programme qui affiche le résultat (écrit en toutes lettres `pile` ou `face`) du lancer d'une pièce truquée de sorte que *pile* apparaît avec la probabilité $\frac{1}{4}$ et *face* avec la probabilité $\frac{3}{4}$ (revoir l'exercice 3.6 du chapitre 3).

Exercice 4.2 (Minimum et maximum d'une liste)
Écrire un programme qui saisit un entier n, puis saisit une liste de n réels (que l'on stockera dans un tableau) et enfin calcule et affiche le minimum et le maximum de ces n réels ainsi que le plus petit rang auxquels ces extrema sont obtenus.

Exercice 4.3 (Équation du second degré)
Écrire un programme qui saisit trois réels a, b et c puis détermine et affiche l'ensemble des solutions de l'équation $ax^2 + bx + c = 0$ dans $\mathbb{R}$. Il faudra tenir compte de toutes les situations possibles (même le cas $a = 0$). On pourra ensuite faire afficher les solutions complexes lorsqu'il y en a (sous la forme : partie réelle puis i puis partie imaginaire).

Exercice 4.4 (Jeu du "c'est plus / c'est moins")
Écrire un programme qui choisit, au hasard, un nombre entier compris entre 1 et 100 puis propose 6 occasions à l'utilisateur de deviner le nombre choisi puis répond après chaque proposition de valeur par l'utilisateur par `gagné`, `plus grand`, `plus petit`. Dans le cas où l'utilisateur n'a pas deviné le nombre, l'ordinateur affiche la phrase : `Le nombre à deviner était ...` (on affiche ici la valeur exacte à deviner). Exemple :

```
Choix 1 : 30
   plus grand
Choix 2 : 70
   plus petit
Choix 3 : 50
   plus petit
Choix 4 : 40
   plus grand
Choix 5 : 45
   plus petit
Choix 6 : 43
```

```
   plus grand
Le nombre à deviner était 44.
```

Chapitre 5

Suites récurrentes, sommes et produits

L'objet de ce chapitre est de découvrir une nouvelle méthode **sans utiliser de tableau** pour calculer des termes successifs d'une suite récurrente ou bien la somme de termes consécutifs d'une suite (récurrente ou non récurrente) ou bien encore le produit de termes consécutifs d'une suite.

5.1 Exemple : la suite des factorielles

On considère la suite définie par : $\forall n \in \mathbb{N},\ u_n = n!$. On transforme cette écriture sous la forme d'une suite récurrente :

$$u_0 = 1 \qquad \text{et} \qquad \forall n \in \mathbb{N}^*,\ u_n = n \times u_{n-1}$$

Recopiez le programme suivant :

```
PROGRAM factorielle ;
VAR i,n,u : integer ;
BEGIN
write('Donner un entier n : ') ; readln(n) ;
u := 1 ;
FOR i := 1 TO n DO u := i*u ;
writeln(n,'! = ',u) ;
readln ;
END.
```

- Testez-le avec $n = 6$ puis avec $n = 8$. Que constatez-vous dans ce dernier cas ? Explication ?
- Quel rapport y a-t-il entre le `i` du programme et le n de la définition de la suite récurrente ?

Remarque :
Si la suite était définie par $u_0 = 1$ et $u_{n+1} = (n+1)u_n$ pour tout entier $n \geqslant 0$, on pourrait remplacer la boucle ci-dessus par la boucle suivante : `FOR i := 0 TO n-1 DO u := (i+1)*u ;`

5.2 Calculs des termes successifs d'une suite récurrente

Théorème 5.1 :
Soit f une fonction numérique définie sur un intervalle I de $\mathbb{R}$ et telle que $f(I) \subset I$. Pour calculer une valeur *approchée* du terme de rang n de la **suite récurrente** u définie par :

$$u_0 = a \in I, \qquad \text{et} \qquad \forall k \in \mathbb{N},\ u_{k+1} = f(u_k)$$

on peut écrire le programme sous la forme suivante :

```
PROGRAM suite_recurrente ;
VAR i,n : integer ;
    u : real ;
BEGIN
write('Donner le premier terme u0 : ') ; readln(u) ;
write('Donner le rang n du terme à calculer : ') ; readln(n) ;
FOR i := 1 TO n DO u := f(u) ;   {remplacer f(u) par ce qu'il faut}
writeln('Le terme u',n,' vaut : ',u) ;
readln ;
END.
```

et l'utilisateur saisit en début de programme la valeur du terme u_0 ainsi que la valeur du rang final n.

Exemples :

Pour chacune des suites, écrire un programme qui saisit son premier terme u_0 et un rang n puis calcule et affiche le terme u_n :

1. $\forall n \in \mathbb{N},\ u_{n+1} = f(u_n)$ avec $f(x) = \dfrac{2x+5}{x+1}$ pour tout réel $x \neq -1$.
2. $\forall n \in \mathbb{N},\ u_{n+1} = u_n e^{-u_n}$.
3. $\forall n \in \mathbb{N},\ u_{n+1} = \sqrt{(n+1)u_n}$.
4. $\forall n \in \mathbb{N},\ u_{n+1} = \dfrac{3u_n + u_n^2}{2^n}$ (utilisez une suite auxiliaire pour le calcul de 2^n).

5.3 Calculs de sommes et de produits

Pour calculer des sommes ou bien des produits, on se ramène à une situation de suite récurrente. Ainsi, si $(u_n)_{n\geqslant n_0}$ est une suite de réels et si $(S_n)_{n\geqslant n_0}$ et $(P_n)_{n\geqslant n_0}$ sont les suites définies par :

$$\forall n \geqslant n_0,\ S_n = \sum_{k=n_0}^{n} u_k \qquad \text{et} \qquad \forall n \geqslant n_0,\ P_n = \prod_{k=n_0}^{n} u_k$$

alors, en posant $S_{n_0-1} = 0$ et $P_{n_0-1} = 1$, on a :

$$\forall n \geqslant n_0,\ S_n = S_{n-1} + u_n \qquad \text{et} \qquad \forall n \geqslant n_0,\ P_n = P_{n-1} \times u_n$$

Un calcul de somme ou de produit fait donc intervenir deux suites simultanément.

Théorème 5.2 (Principe de calcul d'une somme) :
Si la suite $(u_n)_{n\geqslant n_0}$ est définie par la relation de récurrence $u_{n+1} = f(u_n)$ (ou bien $u_{n+1} = f(n, u_n)$) pour tout entier naturel $n \geqslant n_0$ alors le calcul de S_n peut se faire par :

```
S := 0 ; u := ... ; {remplacer ... par le nombre ou l'expression idoine}
FOR i := 1 TO n DO
   BEGIN
   u := f(u) ; {écrire la fonction qui va bien}
   S := S+u ;
   END ;
```

Si la suite $(u_n)_{n\geqslant n_0}$ est définie explicitement $(u_n = f(n))$ pour tout entier naturel $n \geqslant n_0$ alors le calcul de S_n peut se faire par :

```
readln(n) ;
S := 0 ;
FOR i := 1 TO n DO S := S+f(i) ; {écrire la fonction qui va bien}
```

Exemples :

1. Écrire un programme qui calcule $\displaystyle\sum_{k=1}^{n} k^4$ pour un entier n fourni par l'utilisateur.
2. Programmer et tester le calcul de $S_n = \dfrac{6}{\pi^2} \displaystyle\sum_{k=1}^{n} \frac{1}{k^2}$ pour de « grandes » valeurs de n. Interpréter le résultat.
3. Calculer $S_n = \displaystyle\sum_{k=0}^{n} u_k$ avec $u_0 \in \mathbb{R}$ donné par l'utilisateur et $u_{n+1} = u_n^2$ pour tout entier $n \geqslant 0$.
4. Programmer et tester le calcul de $S_n = \displaystyle\sum_{k=0}^{n} \frac{1}{k!}$ pour de « grandes » valeurs de n. Interpréter le résultat.

Théorème 5.3 (Principe de calcul d'un produit) :
Si la suite $(u_n)_{n\geqslant n_0}$ est définie par la relation de récurrence $u_{n+1} = f(u_n)$ pour tout entier naturel $n \geqslant n_0$ alors le calcul de P_n peut se faire par :

```
P := 1 ; u := ... ; {remplacer ... par le nombre ou l'expression idoine}
FOR i := 1 TO n DO
   BEGIN
   u := f(u) ; {écrire la fonction qui va bien}
   P := P*u ;
   END ;
```

Si la suite $(u_n)_{n\geqslant n_0}$ est définie explicitement $(u_n = f(n))$ pour tout entier naturel $n \geqslant n_0$ alors le calcul de S_n peut se faire par :

```
readln(n) ;
P := 1 ;
FOR i := 1 TO n DO P := P*f(i) ; {écrire la fonction qui va bien}
```

Exemples :

1. Écrire un programme qui calcule $\displaystyle\prod_{k=1}^{n}\left(1-\frac{k}{n+1}\right)$ pour un entier $n \geqslant 1$ saisi par l'utilisateur.
   Peut-on exprimer ce produit en fonction de n ? Quelle est sa limite lorsque $n \to +\infty$?
2. Calculer $\left(1+\dfrac{1}{n}\right)^n$ pour $n \in \{10, 100, 1000, 10000\}$.
3. Calculer $\displaystyle\prod_{k=0}^{n} u_k$ pour $u_0 \in \mathbb{R}$ fourni par l'utilisateur et $u_{n+1} = u_n e^{-u_n}$.
4. Écrire un programme qui calcule $0,999^n$ puis $\displaystyle\sum_{k=1}^{n} 0,999^k$ pour un entier n fourni par l'utilisateur.

5.4 Exercices

L'usage des tableaux est évidemment proscrit dans ces exercices.

Exercice 5.1 (Suites récurrentes linéaires d'ordre 2)

Écrire un programme qui saisit les deux premiers termes u_0 et u_1 et un entier $n \geqslant 2$ puis calcule et affiche le terme u_n de la suite définie par :

$$\forall n \in \mathbb{N},\ u_{n+2} = u_{n+1} + u_n$$

Calculer u_{100} avec $u_0 = 0$ et $u_1 = 1$.

Exercice 5.2 (Deux suites imbriquées)
Écrire un programme qui saisit des réels positifs a_0 et b_0, un entier n puis calcule et affiche le terme de rang n de la suite $(a_k)_{k\geqslant 0}$ et celui de la suite $(b_k)_{k\geqslant 0}$ définies par :

$$\forall k \in \mathbb{N},\ a_{k+1} = \frac{a_k + b_k}{2},\ b_{k+1} = \sqrt{a_k b_k}$$

Exercice 5.3 (Sommes doubles)
Écrire un programme qui saisit un entier $n \geqslant 1$ et calcule et affiche les résultats des sommes doubles :

$$\sum_{1\leqslant i<j\leqslant n} \frac{1}{i+j} \qquad \text{et} \qquad \sum_{1\leqslant i\leqslant j\leqslant n} \frac{1}{i+j}$$

Exercice 5.4 (Somme de la série régulièrement alternée)
Écrire un programme calculant la somme :

$$\sum_{k=0}^{n} \frac{(-1)^k}{k+1}$$

pour une valeur de n fournie par l'utilisateur. On testera le programme avec $n = 20000$ et on pourra vérifier que cette somme converge vers $\ln(2)$ lorsque $n \to +\infty$.

Exercice 5.5 (Autres sommes de séries)
Écrire un programme calculant les sommes :

1. $\displaystyle\sum_{k=1}^{n} \frac{x^k}{k}$ pour $x = \frac{1}{2}$ et $n = 10000$,
2. $\displaystyle\sum_{k=0}^{n} \frac{x^k}{k!}$ pour $x = 2$ et $n = 10000$.

Exercice 5.6 (Somme de Riemann, calcul approché d'une intégrale)
Calculer une valeur approchée de l'intégrale :

$$\left(\int_{-5}^{5} e^{-x^2} dx\right)^2$$

par la méthode des rectangles (l'utilisateur donnera le nombre n de rectangles pour effectuer les calculs).

Tester avec différentes valeurs de n (de plus en plus grandes). Qu'observe-t-on ?

Exercice 5.7 (Statistiques)
Que fait le programme qui suit ?

```
PROGRAM Ckoidon ;
VAR i,n : integer ; e,m,u : real ;
BEGIN
write('Donner n : ') ; readln(n) ;
m := 0 ; e := 0 ;
FOR i := 1 TO n DO BEGIN
                   write('Donner un nombre : ') ; readln(u) ;
                   m := m+u ; e := e+u*u ;
                   END ;
m := m/n ; e := sqrt(e/n-m*m) ;
writeln('m = ',m) ; writeln('e = ',e) ;
readln ;
END.
```

Chapitre 6

Itérations « infinies » REPEAT et WHILE

6.1 Exemple

Une urne contient b boules blanches et une boule rouge. On effectue des tirages successifs d'une boule avec remise de la boule tirée entre deux tirages. Le jeu s'arrête dès que l'on tire pour la première fois la boule rouge. On note alors X le numéro du dernier tirage effectué (si toutefois le jeu s'arrête). On veut modéliser cette expérience aléatoire par un programme dans lequel l'utilisateur se contente de fournir l'entier $b \geqslant 1$.

La modélisation choisie ici consiste à noter 0 la boule rouge et à numéroter les boules blanches de 1 à b. Ensuite, on effectue un tirage au sort uniforme sur l'intervalle $[\![0, b]\!]$.

Il est vivement conseillé de tester les deux versions du programme avec de « grandes » valeurs de b (1000 par exemple).

```
PROGRAM tirage_peut_etre_infini_repeat ;
VAR b,X,tirage : integer ;
BEGIN
   REPEAT write('Donner un entier b>=1 : ') ; readln(b) ; UNTIL b>=1 ;
   randomize ; X := 0 ;
   REPEAT tirage := random(b+1) ; X := X+1 ; UNTIL tirage=0 ;
   writeln('X = ',X) ; readln ;
END.
```

La proposition `tirage=0` est un **test d'arrêt** (ou une **condition d'arrêt**) de la boucle `REPEAT...UNTIL`, c'est à dire que la boucle recommence jusqu'à ce que ce test devienne vrai (donc la boucle continue tant que ce test est faux).

```
PROGRAM tirage_peut_etre_infini_while ;
VAR b,X,tirage : integer ;
BEGIN
b := 0 ;
WHILE b<1 DO BEGIN write('Donner un entier b>=1 : ') ; readln(b) ; END ;
randomize ; X := 0 ; tirage := 1 ;
WHILE tirage<>0 DO BEGIN tirage := random(b+1) ; X := X+1 ; END ;
writeln('X = ',X) ; readln ;
END.
```

La proposition `tirage<>0` est appelée **test de continuation** (ou **condition de continuation**) de la boucle `WHILE...DO`, c'est à dire que la boucle recommence tant que ce test est vrai (donc la boucle continue jusqu'à ce que ce test devienne faux).

Remarques (Quelques différences entre les structures) :

- Les deux tests sont évidemment contraires l'un à l'autre dans les deux programmes.
- Le second programme nécessite une initialisation de la variable servant au test d'arrêt puisque le test est effectué avant la première boucle (et donc la boucle peut ne jamais se réaliser, surtout si on oublie l'initialisation dans cet exemple) alors que dans le premier programme la boucle est toujours réalisée au moins une fois puisque le test d'arrêt s'effectue à l'issue de chaque passage.
  Cette différence permet le choix du mode de programmation utilisé en fonction de la situation à modéliser.

6.2 Syntaxe des deux instructions itératives

Nous avons désormais à notre disposition une deuxième technique pour effectuer des boucles, c'est à dire itérer un processus. Cette nouvelle technique permet de répéter une suite d'instructions un nombre de fois qui nous est inconnu au moment d'effectuer la première itération.

Définition 6.1 (Instruction itérative REPEAT ... UNTIL ...) :

```
REPEAT
   instruction_1 ;
   ... {l'une des instructions doit modifier « test_arret »}
   instruction_dernière ;
   UNTIL test_arret ;
```

Remarques :

- À la fin de chaque itération de la boucle, la proposition `test_arret` est soit vraie soit fausse. La boucle s'arrête dès que `test_arret` est évaluée comme vraie sinon le programme effectue une nouvelle itération. Il est donc primordial que la suite d'instructions à itérer modifie la valeur de vérité de `test_arret` sans quoi le programme bouclera indéfiniment [1]. Le programme effectue donc au moins une itération de la boucle (même si `test_arret` est fausse avant la première itération puisque l'évaluation ne se fait qu'à l'issue celle-ci).
- Il est inutile de délimiter le bloc d'instructions à itérer par `BEGIN...END` puisque `BEGIN` (resp. `END`) est « inclus » dans `REPEAT` (resp. `UNTIL`).

La seconde instruction itérative « infinie », à l'instar de `FOR...TO...DO...` et pour les mêmes raisons, présente deux syntaxes différentes selon que la boucle doit itérer une instruction simple ou une instruction composée (c'est à dire une ou plusieurs instructions).

Définition 6.2 (Instruction itérative WHILE ... DO ...) :

```
WHILE test_continuation DO instruction_unique ; {qui doit modifier « test_continuation »}
```

ou bien :

```
WHILE test_continuation DO BEGIN
                           instruction_1 ;
                           ... {l'une des instructions doit modifier « test_continuation »}
                           instruction_dernière ;
                           END ;
```

Remarque :

On réalise une itération de la boucle tant que, en début de chaque boucle, `test_continuation` est vraie et on cesse dès que `test_continuation` devient fausse. Une boucle `WHILE` peut donc ne jamais être exécutée, il suffit pour cela que `test_continuation` soit fausse initialement puisque le test se réalise avant l'exécution de la boucle.

6.3 Exercices

Dans chaque exercice, écrire une version du programme avec `REPEAT` et une autre avec `WHILE`.

Exercice 6.1 (Tirage au sort sans remise)

Modifier le programme donné en exemple dans le cas de tirages successifs sans remise.

Exercice 6.2 (Amélioration du jeu du c'est plus / c'est moins)

Reprendre le jeu de l'exercice 4.4 et remplacer la boucle `FOR`.

Exercice 6.3 (Temps d'attente du k^e pile)

1. Écrire un programme qui simule l'expérience aléatoire dont le résultat est le temps d'attente du k^e *pile* lors de lancers successifs d'une pièce non truquée (l'entier k étant fourni par l'utilisateur).
2. Reprendre la question qui précède avec une pièce truquée dont la probabilité d'apparition de *pile* est de $\frac{1}{4}$ à chaque lancer.

Exercice 6.4 (Résolution d'équation par dichotomie)

On veut résoudre l'équation $f(x) = 0$ par la méthode dite de **dichotomie**. On en rappelle le principe :

- Soit $]a, b[$ un intervalle contenant une unique solution x_0 de cette équation (on a alors $f(a)f(b) < 0$ dans le cas où f est continue sur $[a, b]$ mais ce n'est pas une proposition équivalente).
- Alors la solution x_0 est dans l'intervalle $]a, \frac{a+b}{2}]$ ou bien dans l'intervalle $]\frac{a+b}{2}, b[$ selon que $f(a)f(\frac{a+b}{2}) \leqslant 0$ ou bien que $f(a)f(\frac{a+b}{2}) > 0$.
- On définit donc, par récurrence, les suites $(a_n)_{n\geqslant 0}$ et $(b_n)_{n\geqslant 0}$ par : $a_0 = a$, $b_0 = b$ et pour tout entier naturel n :

$$a_{n+1} = \begin{cases} a_n & \text{si} \quad f(a_n) f\left(\dfrac{a_n + b_n}{2}\right) \leqslant 0 \\ \dfrac{a_n + b_n}{2} & \text{si} \quad f(a_n) f\left(\dfrac{a_n + b_n}{2}\right) > 0 \end{cases}$$

1. Testez toutefois la combinaison de touches [Ctrl]+[Pause] avant de relancer la machine. En tout état de cause, il est primordial de sauvegarder le programme avant de l'exécuter.

$$b_{n+1} = \begin{cases} \dfrac{a_n + b_n}{2} & \text{si} \quad f(a_n)\, f\left(\dfrac{a_n + b_n}{2}\right) \leqslant 0 \\ b_n & \text{si} \quad f(a_n)\, f\left(\dfrac{a_n + b_n}{2}\right) > 0 \end{cases}$$

Ces deux suites sont adjacentes (pourquoi ?), convergent vers x_0 (pourquoi ?) et a_n est une approximation de x_0 à $(b_n - a_n)$ près par défaut (pourquoi ?).

1. Écrire un programme qui :
   (a) saisit les réels a et b (avec $a < b$),
   (b) saisit la précision ε voulue de la solution ($\varepsilon = b_n - a_n$),
   (c) calcule et affiche une valeur approchée à ε près par défaut de la solution, dans l'intervalle $[a, b]$ ainsi que le nombre n d'itérations effectuées.
2. Tester ce programme pour résoudre l'équation $x + \ln(x) = 0$ dans $\mathbb{R}$ puis l'équation $\tan(x) = x$ dans l'intervalle $\left]\frac{\pi}{2}, \frac{3\pi}{2}\right[$.

Exercice 6.5 (Convergence de la série régulièrement alternée)

Écrire un programme qui détermine le plus petit entier naturel n tel que $\left|\ln(2) - \sum_{k=0}^{n} \frac{(-1)^k}{k+1}\right| \leqslant \varepsilon$, pour un réel $\varepsilon > 0$ fourni par l'utilisateur. Tester avec $\varepsilon = 10^{-5}$.

Exercice 6.6 (Limite d'une suite)

Déterminer une valeur approchée à ε près de la limite L de la suite $(u_n)_{n\geqslant 0}$ définie par :

$$u_0 = 0 \qquad \text{et} \qquad \forall n \in \mathbb{N},\ u_{n+1} = \ln\left(1 + e^{-u_n}\right)$$

On considèrera, ici, que u_n est une valeur approchée de L à ε près lorsque $|u_{n+1} - u_n| \leqslant \varepsilon$.

Exercice 6.7 (Autre temps d'attente)

Soit n un entier naturel non nul. On considère n urnes $U_1, \ldots, U_n$, l'urne numéro k comportant k boules blanches et $n+1-k$ boules noires. On choisit une urne au hasard puis on tire successivement et avec remise une boule dans cette urne jusqu'à obtenir pour la première fois une boule blanche.

1. Écrire un programme qui simule cette expérience aléatoire (l'entier n étant fourni par l'utilisateur) et affiche le nombre de tirages effectués.
2. Modifier le programme pour qu'il demande le nombre r de répétitions à effectuer de ce jeu et affiche le nombre moyen de tirages faits lors de ce r répétitions.

Exercice 6.8 (Encore un temps d'attente)

Soit n un entier naturel non nul. On considère une urne contenant n numérotées de 1 à n. On tire successivement et avec remise entre les tirages une boule jusqu'à obtenir pour la première fois tous les numéros possibles.

1. Écrire un programme qui simule cette expérience aléatoire (l'entier n étant fourni par l'utilisateur) et affiche le nombre de tirages effectués (on pourra aussi faire afficher le nombre de tirages de la boule obtenue le plus fréquemment).
2. Modifier le programme pour qu'il demande le nombre r de répétitions à effectuer de ce jeu et affiche le nombre moyen de tirages faits lors de ce r répétitions (on pourra aussi faire afficher le nombre moyen de tirages de la boule obtenue le plus fréquemment lors de chaque répétition).

Chapitre 7

Les fonctions

7.1 Exemples

En plus des fonctions usuelles décrites au chapitre 2, on peut définir de nouvelles **fonctions**.

7.1.1 Avec un paramètre et sans variable locale

Recopier et tester le programme suivant :

```
PROGRAM une_fonction ;
VAR a,b : real ;
FUNCTION f (x : real) : real ;
   BEGIN {de la fonction f}
   f := sqrt(sqrt(1+x)+x) ;
   END ; {de la fonction f}
BEGIN {du programme principal, début de l'exécution quand on clique sur run}
write('Donner un réel : ') ; readln(a) ;
b := f(a) ; {c'est ici que le compilateur fait appel à la fonction f}
writeln('Image de ',a,' par f : ',b) ;
readln ;
END. {du programme principal, fin de l'exécution}
```

7.1.2 Avec un paramètre et des variables locales

Recopier le programme suivant :

```
PROGRAM factorielle ;
VAR n : integer ;
FUNCTION fact (m : integer) : real ;
   VAR i : integer ; produit : real ;
   BEGIN {de la fonction}
   produit := 1 ;
   FOR i := 1 TO m DO prodtuit := i*produit ;
   fact := produit ;
   END ; {de la fonction}
BEGIN {début de l'exécution}
write('Donner un entier n : ') ; readln(n) ;
writeln('La factorielle de n est : ',fact(n)) ;
readln ;
END. {fin de l'exécution}
```

Tester ce programme avec $n = 6$ puis avec $n = 10$.

Remarque :
Afin de pouvoir réutiliser facilement cette fonction, il est conseillé sauvegarder la fonction (et uniquement elle) dans un fichier au nom très explicite (`FACTO.PAS` par exemple).

7.1.3 Plusieurs fonctions

On souhaite afficher la valeur de :

$$\sum_{k=m}^{n} x^k = \begin{cases} x^m \frac{1-x^{n-m+1}}{1-x} & \text{si } x \neq 1 \\ n-m+1 & \text{si } x = 1 \end{cases}$$

pour tout réel x et tout couple d'entiers naturels (m,n) tels que $m \leqslant n$. Recopier (et compléter) le programme suivant :

```
PROGRAM somme_geometrique ;
VAR x,som : real ; m,n : integer ;
FUNCTION puissance (x : real ; k : integer) : real ;
   VAR i : integer ; produit : real ;
   BEGIN {de la fonction puissance}
   produit := ... {à compléter}
   FOR i := ... TO ... DO ... {à compléter}
   puissance := ... {à compléter}
   END ; {de la fonction puissance}
FUNCTION som_geo (x : real ; a,b : integer) : real ;
   BEGIN {de la fonction som_geo}
   IF x=1 THEN som_geo := ... {à compléter}
          ELSE som_geo := ... {à compléter en utilisant la fonction puissance}
   END ; {de la fonction f}
BEGIN {du programme principal}
write('Donner la valeur du réel x :') ; readln(x) ;
write('Donner la valeur du l''entier m : ') ; readln(m) ;
REPEAT
   write('Donner la valeur du l''entier n (>=m) : ') ; readln(n) ;
   UNTIL n>=m ;
som := ... {à compléter} ;
writeln(x:1:3,'^',m,' + ... + ',x:1:3,'^',n,' = ',som) ;
readln ;
END. {du programme principal}
```

7.2 Syntaxe, notion de variables locales ou globales

Définition 7.1 :
Pour déclarer une fonction, il s'agit d'adapter la notation mathématique :

$$\begin{array}{rccc} f & : & E_1 \times \cdots \times E_n & \to & F \\ & & (x_1,\ldots,x_n) & \mapsto & f(x) \end{array}$$

en la notation du langage de programmation Pascal :

```
FUNCTION ma_fonction (parametres : leurs_types) : type_sortie ;
   VAR toutes_les_variables_utiles_localement ;
         {si nécessaire pour des calculs intermédiaires}
   BEGIN
   instructions {si nécessaire pour des calculs intermédiaires}
   ma_fonction := resultat_d_un_calcul_en_fonction_des_parametres ;
   END ;
```

et de placer ceci entre la zone de déclaration de variables (`VAR ...`) et le programme en lui même (`BEGIN ... END.`).

Remarques :

- Une fonction est construite comme un « mini-programme », on dit d'ailleurs que c'est un **sous-programme** :
  - on remplace l'en-tête « `PROGRAM` » par « `FUNCTION` »,
  - on lui donne un nom,
  - on peut déclarer une zone de variables qui sont alors dites **locales**,
  - la fonction en elle-même débute par « `BEGIN` » et s'achève par « `END ;` » (attention, avec un point-virgule et non un point).
- Après la déclaration du nom de la fonction (au sens informatique du terme), on écrit, entre parenthèses, la liste des variables de la fonction. On appelle ces variables les **paramètres** de la fonction. Ils servent à stocker provisoirement les valeurs des données fournies par programme principal. On dit que les paramètres **passent par valeur** (seule la valeur de la variable globale est transmise, pas son nom, c'est à dire son adresse dans la mémoire de l'ordinateur et donc).
- Une restriction importante est à noter à propos des fonctions : l'ensemble d'arrivée ne peut être que `integer` ou `real` ou `boolean` (ce qui ne pose pas de problème pour le moment).
- Une fonction n'est utilisée (par l'ordinateur) que lorsqu'elle est appelée par le programme principal par une instruction du type :

`ma_variable := ma_fonction(parametres)` ou bien `write(ma_fonction(parametres))`

- On peut écrire plusieurs fonctions les unes à la suite des autres, toute fonction pouvant utiliser celles définies auparavant (mais pas celles définies après puisque le compilateur ne les connaît pas encore).
- L'intérêt d'utiliser un sous-programme (et en particulier une fonction) est double :
  - allègement de l'écriture du programme principal,
  - réutilisation dans d'autres programmes (on écrit la fonction une fois pour toute et on la sauvegarde dans un fichier particulier) : pour cela, elle doit être totalement indépendante de tout programme (*self-contained*) c'est à dire qu'elle ne doit jamais utiliser une variable **globale** (définie dans la zone `VAR` qui la précède) ; ainsi une valeur de variable globale nécessaire à la fonction doit être passée par valeur comme paramètre de la fonction et alors la fonction sera réutilisable sans la moindre modification dans n'importe quel autre programme.

Définition 7.2 :
- Une variable définie dans la zone `VAR` située avant les déclarations de fonctions est dite **globale** : elle est utilisable dans toutes les fonctions qui suivent (mais c'est très déconseillé) ainsi que dans le programme principal.
- Une variable définie dans la zone `VAR` située après la déclaration d'une fonction est dite **locale** (à cette fonction) : elle n'est utilisable que dans cette fonction et nulle part ailleurs.

Remarques :
- Une variable locale est une variable qui se crée lors de l'appel de la fonction, elle ne sert que lors de l'exécution de la fonction et elle disparaît lorsque le travail de la fonction s'achève.
- Une variable locale peut porter le même nom qu'une variable globale. Supposons un programme dans lequel sont définies une variable globale `x` et une variable locale `x` dans une fonction `f`.
  - Si le compilateur rencontre la référence `x` lors de l'exécution de la fonction alors c'est la variable locale qui est utilisée (si elle est modifiée, cela ne modifie pas la variable globale homonyme, ce sont deux variables différentes pour la machine, c'est comme s'il existait une variable `x_fct_f` et une variable `x_prog_principal`).
  - Si le compilateur rencontre la référence `x` lors de l'exécution du programme principal, c'est la variable globale qui est utilisée (la variable locale n'existe pas à ce moment-là).

7.3 Exercices

Exercice 7.1 (Schéma de Horner pour l'évaluation d'un polynôme)
Soit $P(X) = a_0 + a_1X + \cdots + a_nX^n$. On a alors :

$$P(X) = a_0 + X\Big(a_1 + X\big(a_2 + X(\dots a_n)\big)\Big)$$

Par exemple, on a :

$$1 + 2X + 3X^2 + 4X^4 = 1 + X\Big(2 + X\big(3 + X(0 + X(4))\big)\Big)$$

Écrire une fonction `horner` qui :
- reçoit en paramètre un réel x,
- saisit un entier n puis saisit successivement les coefficients $a_n, \dots, a_0$ d'un polynôme (en n'utilisant aucun tableau),
- calcule l'évaluation de ce polynôme en le réel x,
- renvoie cette évaluation au programme principal.

Exercice 7.2 (fonctions de plusieurs variables)
Écrire un programme qui saisit trois réels strictement positifs et calcule puis affiche les résultats de :

$$f(x,y,z) = x^y + y^z + z^x \qquad \text{et} \qquad g(x,y,z) = \frac{f(x,y,z)^x + f(x,y,z)^y + f(x,y,z)^z}{xy + yz + zx}$$

La fonction `g` devra utiliser la fonction `f` et devra comporter une variable locale pour stocker le résultat de $f(x,y,z)$.

Exercice 7.3 (nombre de combinaisons, lois binomiale et hypergéométrique)

1. Écrire une fonction `combinaison(n,p)` qui calcule l'entier $\binom{n}{p}$ (nombre de combinaisons à p éléments dans un ensemble à n éléments). Il convient de garder précieusement cette fonction après l'avoir testée. On rappelle que $\binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}$ pour $n \geqslant p \geqslant 1$. Il est vivement déconseillé d'utiliser des factorielles et on ne négligera pas les cas problématiques (comme $n < 0$ par exemple). Enfin, on vérifiera que $\binom{20}{10} = 184756$.
2. Écrire un programme qui saisit un entier n et un réel $p \in]0,1[$ puis complète et affiche un tableau `B` dont l'élément numéro k contient le réel $\mathbb{P}(X = k)$ où $X \hookrightarrow \mathcal{B}(n,p)$. On pourra utiliser des fonctions déjà programmées et sauvegardées.

3. Soit $Y \hookrightarrow \mathcal{H}(N,n,p)$. Écrire un programme qui saisit les deux entiers strictement positifs N et n ($n \leqslant N$) et le réel $p \in]0,1[$, détermine l'ensemble $Y(\Omega)$, complète et affiche un tableau de réels `H` dont l'élément numéro k contient le réel $\mathbb{P}(Y=k)$.
Attention : En Pascal, `N` et `n` désignent la même variable ; le nombre Np doit être un entier.
4. Fusionner les deux programmes précédents en faisant afficher leurs résultats côte à côte (c'est à dire que sur chaque ligne de l'écran doit s'afficher les valeurs de k, $\mathbb{P}(X=k)$ et $\mathbb{P}(Y=k)$. On testera avec :
   (a) $X \hookrightarrow \mathcal{B}\left(10,\frac{1}{2}\right)$ et $Y \hookrightarrow \mathcal{H}\left(20,10,\frac{1}{2}\right)$
   (b) $X \hookrightarrow \mathcal{B}\left(10,\frac{1}{100}\right)$ et $Y \hookrightarrow \mathcal{H}\left(20,10,\frac{1}{100}\right)$
   (c) $X \hookrightarrow \mathcal{B}\left(10,\frac{1}{2}\right)$ et $Y \hookrightarrow \mathcal{H}\left(1000,10,\frac{1}{2}\right)$
   (d) $X \hookrightarrow \mathcal{B}\left(10,\frac{1}{100}\right)$ et $Y \hookrightarrow \mathcal{H}\left(1000,10,\frac{1}{100}\right)$

Exercice 7.4 (moyennes)
Reprendre le programme de calculs de moyennes (exercice 3.3) et le modifier pour qu'il comporte quatre fonctions (nommées `arith`, `geo`, `harmo`, `quad`). On impose que les trois dernières fonctions utilisent la fonction `arith`.

Exercice 7.5 (tableau de valeurs d'une fonction)
À l'aide de deux tableaux (`x` et `y`), écrire un programme qui saisit trois réels `xmin`, `xmax` et `pas` puis complète et affiche sous la forme le tableau de valeurs (sans le cadre) :

x	y
x_{min}	$f(x_{min})$
$x_{min}+pas$	$f(x_{min}+pas)$
...	...
x_{max}	$f(x_{max})$

Tester ce programme avec `xmin=0`, `xmax=10`, `pas=1` ainsi que la fonction définie par : $f(x)=e^{2x}\ln(1+e^{-2x})$.
Ces résultats sont-ils raisonnables ? Commenter.

Exercice 7.6 (tirage au sort uniforme)
Écrire une fonction qui simule le tirage au sort d'un entier choisi au hasard dans $[\![a,b]\!]$ où a et b sont deux entiers fournis par l'utilisateur.

Exercice 7.7 (tirage au sort binomial)
Écrire une fonction qui simule n lancers successifs ($n \geqslant 1$) d'une même pièce dont la probabilité d'apparition de *pile* à chaque lancer est un réel p ($p \in]0,1[$) puis qui retourne le nombre de *pile* obtenus. On pourra utiliser la fonction `random` et interpréter le résultat comme suit : si le résultat est dans $[0,p[$ alors on considèrera que l'on a obtenu *pile* ; si le résultat est dans $[p,1[$ alors on considèrera que l'on a obtenu *face*.

Exercice 7.8 (tirage au sort hypergéométrique)
Écrire une fonction qui simule les n tirages successifs et sans remise d'une boule dans une urne qui contient initialement a boules noires et b boules blanches (les entiers a, b et n sont les paramètres de la fonction) et retourne le nombre de boules noires obtenues.

Exercice 7.9 (encore une variable aléatoire)
Une urne contient r boules rouges, v boules vertes et b boules bleues. On tire une par une et sans remise toutes les boules de cette urne. On note X le rang de tirage tel que, pour la première fois après ce tirage, il ne reste que des boules de seulement deux couleurs différentes dans l'urne. Écrire une fonction qui simule cette expérience aléatoire et retourne la valeur prise par X.

Exercice 7.10 (rang de tirage de la dernière boule rouge)
Une urne contient b boules blanches ($b \geqslant 1$) et r boules rouges ($r \geqslant 1$). On tire successivement et sans remise toutes les boules. On note X la variable aléatoire égale au rang de tirage de la dernière boule rouge.

1. Écrire une fonction `derniere_rouge` dont le résultat est la valeur prise par la variable X à l'issue de cette expérience aléatoire.
2. On répète l'expérience aléatoire précédente 30000 fois. En utilisant la fonction précédente, écrire un programme qui détermine et affiche un tableau des valeurs approchées de la loi de la variable aléatoire X puis une valeur approchée de son espérance et de son écart type.

Exercice 7.11 (loi géométrique et loi de Poisson)
1. Écrire une fonction de paramètres un réel $p \in]0,1[$ et un entier $k \geqslant 1$ qui retourne la valeur de $P(X=k)$ avec $X \hookrightarrow \mathcal{G}(p)$ (on testera avec $p=\frac{1}{6}$) puis avec $X \hookrightarrow \mathcal{P}(\lambda)$ (on testera avec $\lambda=2$).
2. **Comparaison loi binomiale / loi de Poisson.**
Soit $\lambda \in]0,+\infty[$, $n \in \mathbb{N}^*$, $X \hookrightarrow \mathcal{B}\left(n,\frac{\lambda}{n}\right)$ et $Y \hookrightarrow \mathcal{P}(\lambda)$. Écrire un programme qui affiche sur chaque ligne l'entier k et les réels $P(X=k)$ et $P(Y=k)$ pour tous les entiers k tels que $0 \leqslant k \leqslant n$ ou $P(Y=k) > \varepsilon$. On testera le programme avec les données $\lambda=2$, $\varepsilon=10^{-5}$ et $n \in \{10;50;100\}$.

Chapitre 8

Les procédures

8.1 Passage de paramètres par valeur ou par variable

8.1.1 Exemple

```
PROGRAM endomorphisme_de_R3 ;
TYPE vector = ARRAY [1..3] OF real ;
VAR U,V : vector ;
PROCEDURE image_par_f (X : vector ; VAR Y : vector) ;
   BEGIN
   Y[1] := +5*X[1]-3*X[2]-2*X[3] ;
   Y[2] := -3*X[1]-2*X[2]+5*X[3] ;
   Y[3] := -2*X[1]+5*X[2]-3*X[3] ;
   END ;
PROCEDURE saisie_vecteur (VAR X : vector) ;
   VAR i : integer ;
   BEGIN
   FOR i := 1 TO 3 DO BEGIN write('coordonnée ',i,' = ') ; readln(X[i]) ; END ;
   END ;
PROCEDURE affichage_vecteur (X : vector) ;
   VAR i : integer ;
   BEGIN
   FOR i := 1 TO 3 DO writeln('coordonnée ',i,' = ',X[i]) ;
   END ;
BEGIN
writeln('Saisie des coordonnées d''un vecteur de R^3 :') ;
saisie_vecteur(U) ;
image_par_f(U,V) ;
writeln('Vecteur image par f :') ;
affichage_vecteur(V) ;
readln ;
END.
```

8.1.2 Commentaires sur la procédure `image_par_f`

Une fonction (en langage Pascal) ne peut pas donner en guise de résultat un tableau (mais seulement des scalaires : entiers, réels et booléens). Pour contourner ce problème, il existe une instruction nommée `PROCEDURE` qui s'utilise en remplaçant la notation fonctionnelle

```
V := image_par_f(U) ;
```

par la notation procéduriale

```
image_par_f(U,V) ;
```

Lors de la déclaration de cette procédure, il est fondamental de spécifier au compilateur quelles sont les variables de type « données » et quelles sont celles de type « résultats » : les premières ne seront pas modifiées après exécution de la procédure alors que les secondes le seront à coup sûr. On dit alors que le programme principal passe à la procédure des paramètres (ici `U` et `V`) **par valeur** dans le cas de `U` et **par variable** dans le cas de `V` : le contenu de `U` (dans le programme principal) sera inchangé après traitement de son contenu par la procédure alors que le contenu de `V` pourra être modifié. Pour distinguer le passage d'un paramètre par valeur du passage d'un paramètre par variable, on écrit le préfixe `VAR` devant le(s) paramètre(s) à passer par variable d'où l'instruction :

```
PROCEDURE image_par_f (X : vector ; VAR Y : vector) ;
```

Lors de l'appel de la procédure `image_par_f` par le programme principal, le contenu de la variable `U` (resp. `V`) est transféré dans une variable `X` (resp. `Y`). La procédure effectue des calculs puis, à l'issue de son travail, elle ne modifie pas la variable `U` (passage par valeur [1]) mais reporte le contenu de `Y` dans la variable `V` (passage par variable).

Pour des raisons analogues, la procédure `saisie_vecteur` reçoit le paramètre `X` par variable (il y a une saisie donc une modification) alors que la procédure `affichage_vecteur` reçoit le paramètre `X` par valeur (l'affichage ne nécessite pas la modification du contenu).

8.1.3 Exercice

Dans le programme qui suit, on a délibérément omis les indications de passage de paramètre par variable. Recopier le programme en ajoutant les préfixes `VAR` manquant.

Ce programme est un jeu. Il choisit « au hasard » un endomorphisme de $\mathbb{R}^3$ que vous allez devoir deviner. Pour cela, le programme vous demande successivement trois vecteurs de $\mathbb{R}^3$ et vous affiche (au fur et à mesure des saisies) les images de ces vecteurs par l'endomorphisme qu'il a choisi. Ensuite, il choisit « au hasard » un vecteur de $\mathbb{R}^3$ (à coordonnées dans $[\![-10, 10]\!]$) puis vous demande de donner son image par l'endomorphisme (vous n'avez droit qu'à un seul essai).

```
PROGRAM jeu_dans_R3 ;
TYPE vector = ARRAY [1..3] OF integer ;
VAR a : ARRAY [1..9] OF integer ;
    U,V,W : vector ;
    i : integer ;
PROCEDURE saisie_vecteur (X : vector) ;
   VAR i : integer ;
   BEGIN
   FOR i := 1 TO 3 DO BEGIN write('coordonnée ',i,' = ') ; readln(X[i]) ; END ;
   END ;
PROCEDURE affichage_vecteur (X : vector) ;
   VAR i : integer ;
   BEGIN
   FOR i := 1 TO 3 DO writeln('coordonnée ',i,' = ',X[i]) ;
   END ;
FUNCTION vecteurs_egaux (X,Y : vector) : boolean ;
   BEGIN
   vecteurs_egaux := (X[1]=Y[1]) AND (X[2]=Y[2]) AND (X[3]=Y[3]) ;
   END ;
PROCEDURE f (X : vector ; Y : vector) ;
   VAR i : integer ;
   BEGIN
   FOR i := 1 TO 3 DO Y[i] := a[3*i-2]*X[1]+a[3*i-1]*X[2]+a[3*i]*X[3] ;
   END ;
BEGIN
{choix de l'endomorphisme}
   randomize ; FOR i := 1 TO 9 DO a[i] := random(7)-3 ;
{saisie des 3 vecteurs et affichage de leur image}
   FOR i := 1 TO 3 DO
       BEGIN
       writeln('Saisie du vecteur ',i,' :') ; saisie_vecteur(U) ;
       writeln('Son image est :') ; f(U,V) ; affichage_vecteur(V) ;
       END ;
   writeln('Appuyer sur ENTREE') ; readln ;
{le jeu en lui-même}
   for i := 1 TO 3 DO U[i] := random(21)-10 ;
   writeln('Vecteur U tiré au sort :') ; affichage_vecteur(U) ;
   f(U,V) ;
   writeln('Donner f(U) :') ; saisie_vecteur(W) ;
   IF vecteurs_egaux(W,V)
      THEN writeln('Gagné')
      ELSE BEGIN
           writeln('Perdu, le vecteur était :') ;
```

1. Il est possible, si besoin est, de faire modifier le contenu de la variable `X` par la procédure mais la modification ne sera pas reportée dans la variable `U` à la fin.

```
            affichage_vecteur(V) ;
            END ;
   readln ;
END.
```

8.2 Syntaxe des procédures

Définition 8.1 :
Une procédure est un sous-programme que l'on crée pour éviter d'écrire plusieurs fois la même suite d'instructions. Comme pour les fonctions, la déclaration d'une procédure s'effectue après celle les variables globales et avant le programme principal. Elle est de la forme :

```
PROCEDURE nom (données : leur_type ; VAR résultats : leur_type) ;
   VAR variables_locales : leur_type ; {si nécessaire}
   BEGIN
   ...
   END ;
```

Dans le ***programme principal***, l'appel de la procédure s'écrit sous la forme :

```
nom(les_données,les_résultats) ;
```

Remarques :

- Une procédure peut appeler une autre procédure ou une fonction (mais penser à les déclarer dans le bon ordre). Une fonction peut aussi appeler d'autres fonctions ou procédures définies préalablement.
- Il ne faut pas hésiter à créer et à utiliser des variables locales. Une bonne procédure est une procédure « self-contained » : elle n'utilise aucune variable globale du programme sauf celles qu'elle reçoit en paramètres (par valeur ou par variable).
- Pour éviter les confusions, il est conseillé de ne pas nommé les paramètres des procédures avec le nom correspondant dans le programme principal.

8.3 Exercices

Exercice 8.1 (Composées de deux applications linéaires)
Selon le modèle de l'exemple initial, écrire un programme qui saisit un vecteur de $\mathbb{R}^4$ puis détermine et affiche son image par l'application linéaire $g \circ f$ (on ne cherchera surtout pas à déterminer cette application $g \circ f$). On testera le programme avec les applications :

$$f : \begin{array}{ccc} \mathbb{R}^4 & \to & \mathbb{R}^3 \\ \begin{pmatrix} x \\ y \\ z \\ t \end{pmatrix} & \mapsto & \begin{pmatrix} x-y \\ y-z \\ z-t \end{pmatrix} \end{array} \quad \text{et} \quad g : \begin{array}{ccc} \mathbb{R}^3 & \to & \mathbb{R}^2 \\ \begin{pmatrix} x \\ y \\ z \end{pmatrix} & \mapsto & \begin{pmatrix} 2x-y-z \\ -x-y+2z \end{pmatrix} \end{array}$$

Exercice 8.2
Reprendre l'exercice 3.4 sur les permutations des cellules d'un tableau en utilisant des procédures pour la saisie, la permutation et l'affichage des résultats.

Exercice 8.3 (Opérations sur les polynômes)
On définit un nouveau type pour désigner les polynômes (de la même façon que l'on avait défini un nouveau type pour désigner les vecteurs de $\mathbb{R}^3$) :

```
TYPE polynom : ARRAY [0..51] OF integer ;
```

où tous les polynômes utilisés seront à coefficients entiers de degré $\leqslant 50$. Ainsi, si `P` est une variable de type `polynom` alors les variables `P[0],P[1],...,P[50]` contiennent les coefficients de degré 0,1,...,50 du polynôme `P` et *on convient* que la variable `P[51]` contient le degré de `P` (ou -1 si `P` est nul).

1. Écrire des procédures pour saisir un polynôme, afficher un polynôme, ajouter deux polynômes, multiplier un polynôme par un réel, multiplier deux polynômes, déterminer le polynôme dérivé, déterminer le polynôme primitive qui s'annule en 1.
2. Réécrire aussi une fonction Horner.
3. Écrire enfin un programme qui saisit un polynôme P et affiche le polynôme $\left(P' \times \int P\right) - P'(1)[P - P(1)]^2$ où $\int P$ désigne la primitive de P qui s'annule en 1.

Exercice 8.4 (Opérations sur les matrices)

On définit un nouveau type pour désigner des matrices réelles carrées. On utilise pour cela des tableaux à double entrée (extension des tableaux uni-colonnes du langage Pascal) :

```
TYPE matrix : ARRAY [1..10, 1..10] OF real ;
```

Ainsi, si `M` est de type `matrix`, alors `M[2,7]` est la variable réelle qui désigne le coefficient situé en 2ème ligne et 7ème colonne de la matrice `M`.

1. Écrire des procédures pour saisir une matrice, afficher une matrice, ajouter deux matrices, multiplier une matrice par un réel, multiplier deux matrices.
2. Écrire ensuite un programme (utilisant ces procédures) qui saisit deux matrices A et B de $\mathcal{M}_3(\mathbb{R})$ puis qui détermine et affiche la matrice $AB - BA$.