HEB Ecole **S**upérieur d'**I**nformatique

Assistant d'**I**ngénieur 2 ième

Introduction à la programmation.

Pierre BETTENS (PBT)

Ce cours d'introduction à la programmation est destiné aux étudiants de 2 ième AI. Il ne nécessite aucun prérequis si ce n'est l'utilisation d'un ordinateur.

Nous utilisons dans la suite un langage de programmation pour permettre aux étudiants de pouvoir s'entraîner chez eux en écrivant des petits programmes. En effet, notre objectif est double, nous voulons à la fois donner les éléments de base de l'algorithmique aux étudiants et leur permettre l'autoapprentissage ultérieur sur machine.

Ces notes sont dans leur première phase, c'est pourquoi les deux derniers chapitres n'apparaissent pas encore.

Chapitre 1 ALGORITHMIQUE ET PROGRAMMATION

Avant de différencier les notions de **programme** et d'**algorithme**, définissons ce que nous entendons par problème. En effet, un algorithme, comme un programme d'ailleurs, sert à résoudre un problème donné.

1 Notion de problème.

Nous allons nous intéresser à la résolution de **problèmes**. Un exemple de problème concret peut s 'énoncer comme suit ;

« Trouver et imprimer les deux racines de l'équation $x^2-3x+2=0$ »

Résoudre un tel problème consiste en :

Calculer le discriminant $\rho = b^2 - 4ac = 1$

Constater que ce discriminant est positif et donc que les racines sont $x_1 = 1$ et $x_2 = 2$.

C'est l'application stricte de la méthode de résolution d'une équation du second degré. Résoudre un tel problème, en ayant recours à l'informatique, n'a pas beaucoup d'intérêt. Soit on connaît la procédure et le problème revient à imprimer à l'écran les nombres 2 et 1. Soit on ne connaît pas la procédure et le problème est insoluble en ce sens qu'il nous est impossible de générer un processus informatique qui le résoud.

Nous nous appliquerons plutôt à résoudre le **problème abstrait** suivant :

« Trouver et imprimer les deux racines de l'équation $ax^2+bx+c=0$ »

La solution de ce problème, si l'on exclut par simplicité les cas où le discriminant est négatif ou a nul, sont les deux racines suivantes :

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Il est alors aisé de vérifier que notre problème initial se résout en choisissant les valeurs 1,-3 et 2 respectivement pour a,b et c et on pourra écrire un processus informatique résolvant le problème abstrait et donc le problème concret.

La démarche à suivre pour résoudre un problème concret se résume à :

1. **Choisir un problème abstrait** qui englobe le problème initial,
2. **Inventer** (ou trouver dans la littérature) **un algorithme** qui résout le problème abstrait,
3. **Exécuter cet algorithme** avec les données éventuelles du problème.

Le choix du problème abstrait doit se faire de manière judicieuse. Il est clair que ce problème abstrait doit recouvrir un maximum de problèmes concrets mais une trop grande abstraction peut mener à un algorithme trop compliqué voire impossible.

Dans notre exemple, un bon compromis serait un algorithme permettant de résoudre une équation du second degré à coefficients réels.

Soulignons au passage que nous parlons de résoudre des problèmes et non des mystères (Dieu existe-t-il ?). Un problème doit être accessible par la raison.

Pour définir un problème, il faut définir

> L'**objectif à atteindre** ; on désire obtenir les solutions d'une équation du second degré à coéfficients réels.
>
> Le **cadre** dans lequel on travail ; nature des données, …
>
> La **méthode de résolution** ; connaît-on une procédure permettant de résoudre le problème ? Doit-on la chercher ?

2 Algorithme.

L'algorithmique est une science, vieille de 2000 ans et de loin antérieure à la programmation, qui est apparue à la naissance de l'informatique. L'algorithme d'Euclide permettant dans tous les cas de trouver le pgcd de deux nombres est un bel exemple car il était déjà utilisé par les égyptiens.

Mais comment définir la notion d'algorithme ? Par exemple,

Un **algorithme** est une succession d'opérations qui, correctement exécutées, conduit au résultat désiré.

ou encore,

Un **algorithme** est une procédure de résolution contenant des opérations bien définies portant sur des informations, s'exprimant dans une séquence définie sans ambiguïté, et permettant de résoudre un problème ou un ensemble de problèmes de même type.

Définition du Larousse

« Suite de raisonnements ou d'opérations qui fournit la solution de certains problèmes. »

Voici quelque définitions qui permettent de voir que la procédure de résolution d'une équation du second degré est un algorithme. C'est une succession d'opérations élémentaires :

> Calcul du discriminant,
> Détermination de la suite du calcul en fonction de la valeur du discrimionant,
> Calcul des solutions en fonction du choix effectué à l'étape précédente (deux racines complexes, une racine double ou deux racines réelles distinctes)

Une **opération élémentaire** est une opération que l'exécutant (humain ou machine) est capable de réaliser (par exedmple ; additionner deux nombres), soit parce qu'il la connaît soit parce qu'on lui a apprise.

Nous remarquons qu'il existe différents types d'algorithmes. Un algorithme permettant d'approcher la valeur de π est différent de l'algorithme d'Euclide qui calcule le pgcd de deux nombres. Leur différence réside dans le fait que le premier algorithme fournira toujours la même réponse à savoir une valeur approchée de π tandis que le second fournira une réponse différente suivant le couple d'entiers choisis au départ.

Voici deux exemples d'algorithmes classiques.

Algorithme d'Euclide

L'algorithme d'Euclide permet de trouver le pgcd de deux nombres a et b.

Il peut se résumer, si l'on suppose a>b, à ceci :

$$\text{pgcd}\quad(a,\ b) = \begin{cases} \text{pgcd}\quad(b,\ a \bmod b) & \text{si } b \neq 0 \\ a & \text{si } b = 0 \end{cases}$$

Exemple : Vérifier que pgcd (6552,9724) = 52

Recherche dichotomique

Recherche d'un nombre dans une suite triée. Soit la suite

1 3 5 7 9 11 13 16 20

Trouver la position du nombre 16.

Une méthode séquentielle consiste en la comparaison de 1 et 16 ensuite, 3 et 16, 5 et 16 ... Cette méthode n'est pas efficace car dans le pire des cas, elle impose de parcourir toute la suite.

La méthode par dichotomie consiste en une succession de découpage de l'intervalle en deux en ne retenant, à chaque étape, que l'intervalle contenant le nombre.

On compare 16 et 9 (on coupe l'intervalle en deux). 16>9 on conserve le second demi-intervalle. On compare 16 et 13, on conserve le demi-intervalle de droite …

On compare 16 et 16 et c'est terminé.

L'exemple choisi est simple. Essayez de vous convaincre de l'efficacité de l'algorithme en recherchant la position d'un nombre dans une suite de 100 nombres.

VALIDATION DES ALGORITHMES

Face à un algorithme donné, deux questions méritent notre attention :

> Est-il toujours possible de résoudre l'algorithme en un temps fini ? (Le nombre d'opérations à effectuer est-il fini ?)
>
> Le résultat est-il correct ?

Un algorithme sera valable si nous pouvons répondre par l'affirmative à ces deux questions. Répondre à ces deux questions n'est pas toujours simples. Pour s'en convaincre vérifions la validité des algorithmes déjà rencontrés.

Résolution d'une équation d'u second degré.

Nous vérifierons l'algorithme dans le cas d'un discriminant strictement positif, les deux autres cas seront laissé à la discrètion du lecteur.

Pour ce faire, il suffit de vérifier que les équations suivantes s'impliquent l'une l'autre.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
$$2ax + b = \pm\sqrt{b^2 - 4ac}$$
$$(2ax + b)^2 = b^2 - 4ac$$
$$ax^2 + bx + c = 0$$

Algorithme d'Euclide.

La vérification de l'algorithme d'Euclide repose sur son théorème.

La démonstration du théorème repose sur le fait qu'une suite strictement décroissante d'entiers positifs accepte un minorant nul. Si l'on vérifie égalemnt que *pgcd (a,b) = pgcd(b,a mod b)* la validité du théorème est montrée.

Dans la suite du cours, nous ne nous intéresserons pas aux problèmes de terminaison des algorithmes ni à la démonstration de leur justesse. Ces problèmes sortent du cadre d'un cours d'introduction à la programmation.

3 Programme.

Un **programme** est la résolution, par la machine, d'un algorithme. C'est donc une suite d'opérations élémentaires (primitives), exécutables par la machine, qui conduit au résultat cherché.

Pour ce faire, un programme doit être compréhensible par la machine. Pour que le processeur soit à même d'exécuter sa tâche, il faut qu'il puisse :

Comprendre ce que chaque étape signifie

Exécuter l'opération correspondante.

Un programme sera écrit dans un certain **langage de programmation** et se compose d'un ensemble structuré d'instructions (connues par la machine), chacune d 'elles spécifiant les opérations à exécuter par la machine. C'est la seule manière de faire exécuter, par la machine, un algorithme donné.

La programmation n'est cependant pas une simple démarche de traduction car il faut que le programme soit écrit en accord avec les règles de sémantique et de syntaxe du langage utilisé. Dans ce cas, le programme sera compréhensible par la machine … encore faut-il qu'il produise les résultats escomptés. L'activité de programmation consiste à générer une suite structurée d'instructions produisant les mêmes résultats que ceux obtenus par un processeur humain réalisant l'algorithme.

Chapitre 2 LANGAGE DE PROGRAMMATION.

Un programme est écrit dans un certain langage de programmation. Etudions le chemin qui mène du processeur au programmeur. Nous nous permettons cette légère disgression pour permettre au lecteur de situer la notion de programmation dans son ensemble.

1 Langage machine.

Comme nous l'avons dit, un ordinateur est une machine qui peut résoudre des problèmes en exécutant une suite d'instructions qui lui sont connues. Un ordinateur est composé de circuits électroniques et ces circuits sont traversés ou pas par un courant. Raisonnement quelque peu réducteur menant à la conclusion qu'un ordinateur ne peut reconnaître qu'un ensemble d'instructions simples qui finalement se résument à un ensemble de 0 et de 1. Tous les programmes doivent donc être traduits en termes de ces instructions pour pouvoir être exécutés. Ces instructions simples sont du type :

Additionner deux nombres
Vérifier si un nombre est nul
Copier une information d'un endroit de la mémoire vers un autre.

L'ensemble de ces instructions élémentaires forme un langage, compréhensible par la machine, que l'on appelle le **langage machine**. Ce langage n'étant qu'une suite de nombres, il est peu productif pour un humain de l'utiliser et l'activité du programmeur ne se situe pas à ce niveau.

2 Enrichissement des langages.

Le langage machine étant difficilement utilisable par l'homme, on crée un autre ensemble d'instructions plus compréhensibles. Ce nouvel ensemble d'instructions forme aussi un langage, appelons-le L2. L1 sera le langage machine.

Les programmes seront donc écrits en L2 pour être exécutés par la machine en L1. Comment faire ? Il y a deux manières de procéder.

La **première** consiste à remplacer chaque instruction de L2 par une séquence d'instructions de L1 ayant un effet équivalent. Nous aurons ainsi un programme équivalent écrit en L1. L'ordinateur exécutera le programme en L1, nous dirons qu'il y a eu **traduction** du programme ou bien que le programme a été **compilé**.

La **deuxième** manière de procéder est l'utilisation d'un programme ; l'**interprèteur** qui en prenant le programme en L2 comme donnée, l'exécute instruction par instruction. Pour

chaque instruction l'interprèteur exécute une suite d'instructions en L1. Dans ce cas on ne traduit pas tout le programme avant, cela se fait au fur et à mesure à l'aide de l'interprèteur.

Dans les deux cas, l'utilisateur pourra utiliser le langage L2 sans trop se soucier de ce que fait la machine. La machine, quant-à elle, exécutera des instructions en L1 que l'utilisateur ne verra pas.

S'il s'avère que le langage L2 est encore trop lointain du mode de pensée humain (ce qui est le cas), on inventera un langage L3 qui sera traduit en L2 ou exécuté par un interprète écrit en L2.

Si le langage L3 est encore jugé trop primitif, on inventera un langage L4 et ainsi de suite …

On pourra considérer une machine comme une série de couches ou de niveaux de plus en plus « évolués ». L'utilisateur ne s'intéressera qu'aux niveaux les plus évolués. Nous allons rapidement passer quelques niveaux en revue.

NIVEAU 1-2-3

Les trois premiers niveaux sont interprêtés, ils sont composés de langages numériques.

Le niveau 1 ne comprend, en général, qu'une trentaine d'instructions de copies et de tests simples. C'est le niveau le plus bas, les instructions seront directement effectuées par la machine. C'est le **niveau microprogrammation**.

Le niveau 2 est le **niveau machine conventionnelle**.

Le niveau 3 est le **niveau système d'exploitation**.

C'est trois niveaux ne servent qu'a supporter les niveaux supérieurs. Les interprèteurs et traducteurs écrits à ces niveaux le sont par des programmeurs système.

NIVEAU 4

Ce niveau est celui du **langage d'assemblage.** Le langage d'assemblage est le langage de plus bas niveau comprenant des signes intelligibles pour l'homme. Il contient des mnémoniques permettant essentiellement des copies, des comparaisons d'informations.

Nous reviendrons plus tard sur le langage d'assemblage.

NIVEAU 5

Ce niveau est celui utilisé par les programmeurs. C'est à ce niveau que nous allons nous intéresser durant ce cours. C'est le niveau des **langages de haut niveau** tels que FORTRAN, PASCAL, COBOL, BASIC, C, PROLOG, …

Les programmes écrits dans ces langages sont en général traduit par un programme appelé **compilateur**.

NIVEAU 6

Le niveau de l'utilisateur, celui que chacun peut voir, c'est celui d'Excel, de Word d'une console PlayStation … C'est le niveau **langage de l'application**. C'est à ce niveau que l'informatique semble simple de par sa facilité d'utilisation.

Chapitre 3 PASCAL, LANGAGE DE HAUT NIVEAU

Nous allons dans ce chapitre définir les concepts essentiels de la programmation. Nous utiliserons pour ce faire la structure de langage PASCAL. Ce langage est un langage structuré qui permet donc la modularité.

1 Notions de bases

Le premier programme que l'on rédige habituellement permet d'écrire à l'écran un message de bienvenue. Ce qui donne,

```
program programme_1(output);
begin
  Writeln('Bonjour');
end.
```

Writeln est une primitive du langage qui permet d'écrire quelque chose à l'écran et d'aller à la ligne. Les primitives d'entrée/sortie du langage sont **write, writeln, read, readln** que nous verrons plus en détail au fur et à mesure du cours.

On pourra bien sur écrire d'autres choses, par exemple la valeur de $\sqrt{2}$. On utilisera alors une primitive supplémentaire **sqrt** qui calcule la racine carrée d'un nombre. La fonction sqrt fait partie de la librairie standard de PASCAL qui met à disposition à certain nombre d'instructions.

```
program programme_2(output);
begin
  Writeln(sqrt(2));
end.
```

SYMBOLES

Un programme est une suite de mots qui n'ont pas tous la même signification. Dans le programme précédent, il est clair que le mot « begin » n'est pas du même genre que « programme_2 ».

Nous utiliserons des symboles n'ayant pas tous la même signification (+, }, a …) et qui, suivant leur utilisation, représenteront différentes choses. On peut distinguer

les **identificateurs standards**; ce sont les mots propres au langage, ils ont une signification particulière et ne peuvent en changer (begin, end, if, integer, repeat, program, + …

les **identificateurs** ; nous pouvons définir nous même des objets par une suite de symboles qui ne forme pas un identificateur standard (x, jean, nombre_1 …)

les **opérateurs primitifs** ; +, -, *, DIV, MOD, < , >, ET, = …

le **symbole d'affectation** ; := (ne pas confondre avec le =)

VARIABLES ET CONSTANTES

Dans la structure d'un programme écrit en PASCAL, on déclare les variables et les constantes en début de programme.

Une variable permet de mémoriser une information dont le nom et le type restent inchangés durant tout le programme. Les principaux types sont ; **integer**, **real**, **boolean** (vrai ou faux), **char** (caractère). Exemple de déclaration de variables :

```
var
    a : integer ;
    x, y, z : real ;
    nom, prénom : char ;
```

Il y a deux manières de donner une valeur à une variable.

La *première*, interne, se fait par le biais de l'**instruction d'affectation**. On trouvera, par exemple, dans le corps du programme des instuctions de la forme ;

```
a := 12 ;
y := 4.666 ;
x :=(5*y)/7 ;
prénom := 'Jean' ;
```

Dans le membre de gauche se trouve le nom de la variable et dans le membre de droite la valeur que l'on veut lui attribuer. Attention, ces valeurs doivent être du même type.

La *seconde* manière de faire, externe, consiste en l'utilisation de la primitive **read** qui permet l'attribution du valeur donnée par l'utilisateur. Exemple ;

```
read (a) ;
```

Une variable peut garder la même valeur tout au long du programme sans que l'on ait la possibilité de la modifier, on parle alors d'une **constante**. On définit une constante pour les valeur fixe telles que π ou pour une valeur que l'on retrouve à plusieurs endroits dans le

programme et « susceptible de changer ». Par exemple un taux de change, une constante nombres de lignes par page … Exemples

```
const
    e = 2,71 ;
    tva = 0,21 ;
    nombredepages = 40 ;
    blanc = ‘ ‘ ;
```

Avec cette notion de variable, on peut améliorer le programme 2 et écrire un programme qui lira un nombre à l’écran et après calcul de sa racine carrée, l’affiche à l’écran. Ce qui donne ;

```
program programme_3 (input,output) ;
var
    x : real ;
begin
    readln (x) ;
    writeln (sqrt(x)) ;
end.
```

MODULES

En règle générale en PASCAL, une instruction se termine toujours par le symbole **;** sauf si elle est suivie d’un **end**. Lorsque l’on a un ensemble de plusieurs instructions, elles seront toujours précédées d’un **begin** et suivie d’un **end**. Qui marquent, respectivement, le début et la fin d’un bloc ou module.

Nous sommes maintenant capables d’écrire des programmes simples permettants d’effectuer des calculs élémentaires.

Exercices :

1. Ecrire un programme qui lit deux nombres, calcule leur somme et affiche le résultat.
2. Ecrire un programme qui calcule l’aire d’un cercle connaissant son rayon.
3. Ecrire un programme qui, pour la donnée d’un temps exprimé en secondes, le traduit en heure, minutes et secondes.

2 Structures alternatives.

STRUCTURE IF-THEN

```
if condition
    then
        instruction ou séquence d'instructions
    ;
```

La *condition* fournit un résultat logique. Si le résultat est vrai alors l'instruction ou séquence d'instructions est effectuée sinon non et le programme passe à l'instruction suivante.

Exemple

1. Dans une entreprise, une retenue spéciale de 15% est effectuée sur la tranche de salaire supérieure à 40 000 Fb. Etant donné un salaire, calculer le salaire net.

```
program program_4 (inpout, output) ;
const
    Bareme = 40000 ;
    Taux = 0.15 ;
var
    Salaire, Retrait, Net : real ;
begin
    Retrait := 0 ;
    write ('Quel est votre salaire ?') ;
    readln (Salaire) ;
    if Salaire > Bareme then
        Retrait := Taux * (Salaire-Bareme) ;
    Net := Salaire – Retrait ;
    writeln('Votre salaire net est de ',Net) ;
end.
```

2. Soit la donnée de deux nombres entiers, les ordonner et les afficher.

```
program program_5 (inpout,output) ;
var
    a,b,tempo :integer ;
begin
    write ('Entre deux nombres :  ') ;
     read (a,b) ;
     if a>b then
        begin
        tempo :=a ;
        a :=b ;
        b :=tempo
        end ;
    writeln('Le nombre ',a,' est plus petit que ',b) ;
end.
```

STRUCTURE IF-THEN-ELSE

```
if condition
    then
        instruction ou séquence d'instructions 1
    else
        instruction ou séquence d'instructions 2
    ;
```

La *condition* fournit un résultat logique. Si cette condition est vraie alors, l'instruction ou la séquence d'instructions 1 est effectuée, sinon, c'est l'instruction ou la suite d'instructions 2 qui l'est.

Si le programme précédent ne demandait que d'afficher les deux nombres dans l'ordre, on aurait pu écrire ;

```
program program_6 (inpout,output) ;
var
    a,b :integer ;
begin
    write ('Entre deux nombres :  ') ;
     read (a,b) ;
     if a>b then
        writeln('Le nombre ',b,' est plus petit que ',a)
        else
        writeln('Le nombre ',a,' est plus petit que ',b) ;
end.
```

Les instructions **if-then** et **if-then-else** peuvent être imbriquées. Dans ce cas, on veillera à utiliser les tabulations avec rigueur de manière à garder un programme lisible.

Supposons que pour un nombre N donné, nous voulions savoir s'il est divisible par 5 lorsqu'il est impair et divisible par 7 lorsqu'il est pair. Une solution serait ;

```
program programme_7 (input,output)
var
    n : integer;
begin
    write('Entre un entier: ');
    readln(n);
    if (n mod 2)=0 then
        if (n mod 7)=0 then
            writeln(n,' pair est divisible par 7')
            else
            writeln(n,' pair n''est pas divisible par 7')
        else
        if (n mod 5)=0 then
            writeln(n,' impair est divisible par 5')
            else
            writeln(n,' impaire n''est pas divisible par 5');
end.
```

STRUCTURE CASE OF

Cette instruction est une extention de l'instruction **if**. Elle permet de choisir une parmi plusieurs actions en s'orientant suivant les différentes valeurs d'un scalaire.

```
case variable of
    liste 1 de valeurs : instruction ou séquence d'instructions ;
    liste 2 de valeurs : instruction ou séquence d'instructions ;
    liste 3 de valeurs : instruction ou séquence d'instructions ;
    .
    .
    liste n de valeurs : instruction ou séquence d'instructions ;
end
```

La valeur de la variable doit se trouver dans une des listes sinon, le programme génère une erreur. Signalons que certains compilateurs prévoient une instruction **otherwise** qui permet de regrouper les valeurs de la variable qui n'ont pas été mentionnées auparavant. Cette étiquette otherwise doit se placer à la suite de toutes les autres. L'instruction aura la forme suivante ;

```
case variable of
    liste 1 de valeurs : instruction ou séquence d'instructions ;
    liste 2 de valeurs : instruction ou séquence d'instructions ;
    liste 3 de valeurs : instruction ou séquence d'instructions ;
    .
    .
    liste n de valeurs : instruction ou séquence d'instructions ;
    otherwise          : instruction ou séquence d'instructions ;
end
```

Pour illustrer l'emploi de cette structure supposons que l'on veuille écrire un programme gérant les grades attribués en fonction d'une cote sur 20. L'emploi d'un case of sera plus judicieux qu'une imbrication de if.

```
program programme_8 (input,output) ;
var
    c : integer;
begin
    write('Entrer la cote: ');
    readln(c);
    case c of
            1,2,3,4,5,6,7,8,9 : writeln('Vous n''êtes pas reçu.');
            10,11 : writeln('Réussite.');
            12,13 : writeln('Satisfaction.');
            14,15 : writeln('Distinction.');
            16,17 : writeln('Grande distinction.');
            18,19 : writeln('La plus grande distinction.');
            20    : writeln('La plus grande distinction avec félicitations du jury.')
    end
end.
```

Exercice résolu

Résolution d'une équation du second degré à coéfficients réels. Pour résoudre un tel exercice, nous devons traiter les différents cas suivant la valeur du discriminant mais également les différents cas dégénérés provenant de la nullité de certains coefficients. La solution est la suivante.

```
program second_degre (inpout, output) ;
var
    a,b,c,rho,re,im : real;
begin
    read(a,b,c);
     if a=0 then
            if b=0 then
                    writeln('L''équation est dégénérée: ',c,'=0')
                    else
                    begin
                    writeln ('C''est une équation de premier degré.');
                    writeln ('La solution est ',-c/b)
                    end
            else if c=0 then
                    writeln('Les racines sont ',-b/a,' et 0')
                    else
                    begin
                    rho:=sqr(b)-4*a*c;
                    re:=-b/(2*a);
                    im:=sqrt(abs(rho))/(2*a);
                    if rho >=0 then
                            writeln('Les racines sont ',re+im,' et ',re-im)
                            else
                            writeln('Les racines complexes sont ',re,'+',im,'i et ',re,'-',im,'i');
                     end;
end.
```

Exercices

1. Soit une famille de N personnes. Ecrire un programme qui précise si c'est une famille nombreuse ou non (famille nombreuses à partir de 3 enfants).
2. Etant donné un nombre, écrire un programme qui précise si ce nombre est positif, négatif ou nul.
3. Etant donnés deux nombres, écrire un programme qui affiche le plus petit.
4. Ecrire un programme qui donne le maximum de trois nombres N1, N2, N3.
5. Etant donné le solde du compte en banque d'un client et le montant qu'il désire retirer, écrire un programme permettant de donner les résultats suivants sachant que la banque n'accepte pas de solde négatif inférieur à -25 000 :

   retrait autorisé
   retrait partiel autorisé
   retrait refusé

6. Ecrire trois nombres donné dans l'ordre croissant.
7. Etant donné un prix hors tva et un code tva (t1=6%, t2=12% et t3=21%), écrire un programme qui donne le prix tvac.

3 Structures itératives

Les structures itératives sont utilisées lorsque l'on veut faire s'éxécuter un certains nombres de fois, connu ou non, une instruction ou séquence d'instructions.

STRUCTURE WHILE-DO

La première structure itérative, peut-être la plus usitée, éxécute l'instruction ou la séquence d'instructions tant qu'une condition est vraie. Il faudra donc veiller à ce que l'instruction ou la séquence d'instructions modifie une des composantes de la condition pour que celle-ci puisse devenir fausse à un moment donné. Dans le cas contraire, on aurait une boucle infinie !

```
while condition do
    instruction ou séquence d'instructions
    ;
```

Le processus s'effectue comme suit ; évaluation de la condition, exécution de l'instruction ou la séquence d'instructions si la condition est vraie, si elle est fausse, le processeur passe à l'instruction suivante. Nous savons donc qu'après une instruction while, sa condition est fausse. Remarquons également que si la condition est fausse dès le départ, l'instruction ou séquence d'instructions n'est pas éxécutée.

Pour illustrer, écrivons un programme qui affiche un compte à rebours à partir d'un entier n donné.

```
program programme_9 (input,output) ;
var
    n : integer;
begin
    write('Entrer l''entier: ');
    readln(n);
    while n>0 do
            begin
            write(' ',n);
            n:=n-1
            end;
    write('  boum')
end.
```

STRUCTURE REPEAT-UNTIL

Cette structure a quelques différences fondamentales par rapport à la précédente.

Elle va exécuter une instruction ou séquence d'instructions **jusqu'à ce qu'** une condition devienne vraie. On suppose la condition fausse au départ, et on attend qu'elle devienne vraie. Une des composante de la condition doit donc être présente dans la suite d'instructions pour que la boucle puisse s'arrêter.

A l'inverse de la structure while, l'instruction ou séquence d'instructions sera d'office effectuée une fois étant donné que la condition est testée à la fin de la boucle.

La forme générale de cette structure est la suivante ;

```
repeat
    instruction 1 ;
    instruction 2 ;
    instruction 3 ;
    .
    .
    instruction  n;
until condition ;
```

Nous pouvons réécrire l'exemple du compte à rebours en utilisant cette structure.

```
program programme_10 (input,output) ;
var
    n : integer ;
begin
    write('Entre un nombre') ;
    readln(n) ;
    repeat
        write(n,' ') ;
        n :=n-1
    until n=0 ;
end.
```

STRUCTURE FOR-DO

Lorsque le nombre de répétition de la boucle est préalablement connu, on utilise la structure for-do. Sa forme générale est la suivante ;

for variable de contrôle := val_début **to** val_fin **do**
instruction ou séquence d'instructions **;**

C'est la structure qui modifie la valeur de la variable de contrôle. Cette variable de contrôle ne peut pas être modifiée dans le corps de la boucle. A la sortie de cette boucle, sa valeur est *indéterminée*. Le compilateur ne teste pas la condition de sortie à chaque éxécution du corps de la boucle. C'est pourquoi nous conseillons l'emploi d'un for-do plutôt que d'un while chaque fois que cela est possible.

Il existe une instruction **for-downto-do** qui, elle, décrémente la variable de val_début jusqu'à val_fin. Nous pouvons une dernière fois réécrire le programme affichant un compte à rebours.

```
program programme_11 (input,output) ;
var
    n,i : integer;
begin
    write('Entrer l''entier: ');
    readln(n);
    for i:=n downto 0 do
            write (' ',i);
end.
```

Exercice résolu

Calcul d'une racine carrée par la méthode de Newton (sans utiliser sqrt).

Nous utiliserons la méthode de Newton pour calculer, par approximations successives, la racine carrée d'un nombre x.

Si *app* est une approximation de $\sqrt{x}$ alors $\frac{\frac{x}{app} + app}{2}$ est une meilleure approximation.

Le processus itératif débute avec la valeur 1 et se poursuit jusqu'à l'obtention d'une précission suffisante. Nous utiliserons le critère d'arrêt suivant ;

$$\left| \frac{x}{app^{2}} - 1 \right| < 10^{-6}$$

On obtiendra comme programme ;

```
program racinecarree(input,output);
const
    epsilon = 1E-6;
var
    x,racine : real;
begin
    repeat
    write('Entre un entier: ');
    readln(x);
    if x <0 then
            writeln('Le nombre ',x,' n''a pas de racine carrée réelle')
            else if x=0 then
            writeln('La racine carrée de ',x,' est 0')
            else
            begin
            racine := 1;
            repeat
               racine:=(x/racine+racine)/2
            until abs(x/sqr(racine)-1)<epsilon ;
            writeln('La racine carrée de ',x,' est ',racine)
            end;
     until x=0
end.
```

Exercice

1. Pour un entier n donné, énumérer la suite croissante des n premiers nombres pairs.
2. Générer les p premiers termes de la table de multiplication par n.
3. Enumérer les n premiers nombres de la suite de Fibonacci (F_0=0,F_1=1, .. $F_n = F_{n-1} + F_{n-2}$)
4. Calculer factorielle de n.
5. Effectuer la division entière de deux nombres sans utiliser la multiplication ni la division. Afficher le quotient et le reste.
6. Calculer la somme des chiffres qui composes un nombre. Utiliser le résultat pour déterminer si un nombre donné est divisible par 3.
7. Ecrire un programme qui affiche la représentation binaire d'un nombre donné en base 10. (Une seule opération de lecture et d'écriture)
8. Ecrire un programme qui affiche la représentation octale d'un nombre donné en base 10.
9. Calculer le nombre de termes de la série harmonique ($\sum_{i=1}^{n} \frac{1}{i}$) nécessaire pour atteindre une limite donnée.
10. Calculer le PPCM de deux entiers donnés.

11. Ecrire un programme qui permet de jouer au jeu de la fourchette. Il s'agit de trouver un nombre compris entre 1 et 100 en huit essais maximun. L'ordinateur signale si le nombre proposé est trop petit, trop grand ou égal (c'est gagné) et quand c'est nécessaire perdu (lorsqu'il y a plus de huit essais infructueux).
12. Ecrire un programme qui calcule la racine cubique d'un nombre *n* en utilisant la récurrence suivante (on partira d'un $x_0=1$);

$$x_k = \frac{2\,x_{k-1} + \frac{n}{x_{k-1}^2}}{3}$$

13. Ecrire un programme qui exprime toute somme d'argent comprise entre 0 et 1000 francs en nombre de pièces de 50, 20, 5 et 1 francs.

4 Procédures et fonctions.

La bonne manière de résoudre un problème compliqué est de le décomposer en problèmes plus simples. De plus, il est aisé de se rendre compte qu'un long programme est difficile à lire c'est pourquoi, il est conseillé d'utiliser au maximum les **procédures** et les **fonctions**. Une procédure ou une fonction est tout simplement un module logique auquel on peut faire référence par son nom.

PROCEDURES.

Supposons que nous voulions écrire un programme qui lise un texte en sautant les blancs. Nous aurions besoin d'une instruction qui permet de sauter les blancs. Et donc, dans le corps du programme on retrouverait une instruction du type ;

```
repeat
    read(car)
until car <> blanc
```

Au lieu de rencontrer cette instruction dans le corps du programme, on pourrait nommer cette séquence *nom_quelconque* et chaque fois que l'on en a besoin, on l'appelle par son nom. On retrouvera dans le corps du programme l'appel de procédure suivant ;

```
nom_quelconque ;
```

Il n'empêche qu'il faudra définir cette procédure *nom_quelconque* quelque part. Cela se fera comme suit ;

```
procedure nom_quelconque ;
    begin
    repeat
        read(car)
    until car <> blanc
    end ;
```

Nous allons voir sur un exemple comment utiliser les procédures en PASCAL. Ecrivons un programme qui calcule les sommes partielles d'une série harmonique. Le programme doit évaluer *H(n)* pour différentes valeurs de *n*.

$$H\ (n)\ =\ 1\ +\ \frac{1}{2}\ +\ \frac{1}{3}\ +\ \cdots\ +\ \frac{1}{n}$$

Le résultat peut-être exprimé à l'aide d'un nombre rationnel que l'on pourra simplifier. ce nombre rationnel sera représenté par deux variables entières. Pour la simplification, il faut supprimer les facteurs communs au numérateur et au dénominateur et, pour ce faire, rechercher le pgcd des deux nombres. Nous utiliserons l'algorithme d'Euclide. Remarquons que nous supposerons que le numérateur est supérieur au dénominateur ce qui est bien le cas dans la cas de la série harmonique.

```
procedure reduction ;
    begin
        n :=numerateur ;
        d :=denominateur ;
        while d<>0 do
            begin
            reste :=n mod d ;
            n :=d ;
            d :=reste ;
            end ;
        if n>1 then
            begin
            numerateur :=numerateur div n ;
            denominateur :=denominateur div n
            end
    end
```

Nous avons défini une procédure qui ne peut être exécutable seule. Voici un programme qui réduit un nombre fractionnel plus grand que 1 (cette vérification n'est pas faite).

```
program simplification (input,output) ;
var
    numerateur,denominateur : integer ;
    n,d,reste : integer ;

procedure reduction ;
    begin
        n :=numerateur ;
        d :=denominateur ;
        while d<>0 do
            begin
            reste :=n mod d ;
            n :=d ;
            d :=reste ;
```

```
                end ;
            if n>1 then
                begin
                numerateur :=numerateur div n ;
                denominateur :=denominateur div n
                end
    end ;

(* Corps du programme *)
begin
    write('Entre un nombre rationnel supérieur à 1') ;
    readln(numerateur,denominateur) ;
    reduction ;
    writeln(numerateur,' / ',denominateur)
end.
```

Ce programme appelle quelques remarques. La structure générale d'un programme PASCAL est toujours de la forme

Entête du programme
Déclaration des constantes
Déclaration des types
Déclaration de variables
Déclaration et définition des procédures
Corps du programme

Le programme se déroulera comme suit ; les instructions write et read, suivie de l'appel de procédure et donc de son exécution, retour après l'appel et exécution de la dernière instruction writeln.

Vous remarquez également l'apparition de *commentaires*. Il est toujours important d'écrire un minimum de commentaires pour faciliter la relecture d'un programme. Les premiers programmes que nous avons écrits, de par leur simplicité, était facilement compréhensibles ce qui explique l'absence de commentaires.

Les variables *n, d* et *reste* sont déclarées dans la zone de déclaration de variables du programme or qu'elles ne sont utilisées que dans la procédure. Ces variables, ainsi que *numerateur* et *dénominateur* sont des **variables globales.** Elles sont définies et connues dans tout le programme ce qui est inutile ici.

On préfèrera le programme suivant.

```
program simplification (input,output) ;
var
    numerateur,denominateur : integer ;

procedure reduction ;
    var
        n,d,reste : integer ;
    begin
        …
    end ;

(* Corps du programme *)
begin
    write (‘Entre un nombre rationnel supérieur à 1’) ;
    readln (numerateur,denominateur) ;
    reduction ;
    writeln(numerateur,’ / ‘,denominateur)
end.
```

Les variables *n, d* et *reste* sont maintenant des **variables locales** c’est-à-dire qu’elles ne sont connues qu’à l’intérieur de la procédure. Elles sont réservées à l’entrée de celle-ci et « détruites » à la sortie. Lorsqu’il y a plusieurs appels de la même procédure, au deuxième appel les valeurs contenues dans ces variables ne sont pas celles qu’il y avait à la sortie du premier appel.

Lorsque l’on appelle la procédure *réduction*, on utilise les variables globales *numérateur* et *dénominateur*. Si l’on voulait réduire plusieurs nombres rationnels en utilisant la même procédure on devrait à chaque fois écrire le numérateur dans la variable *numerateur* et le dénominateur dans la variable *denominateur*. Supposons que l’on lise 3 nombres, cela donnerait par exemple;

```
…
read(num1,den1) ;
read(num2,den2) ;
read(num3,den3) ;
numerateur :=num1 ;
denominateur :=den1 ;
reduction ;
numerateur :=num2 ;
denominateur :=den2 ;
reduction ;
numerateur :=num3 ;
denominateur :=den3 ;
reduction ;
…
```

On préfèrera donner des arguments à la procédure et l'appel de la procédure se fera comme suit ;

```
reduction (num1,den1) ;
reduction (num2,den2) ;
reduction (num3,den3) ;
```

Il faudra modifier la déclaration de la procédure pour lui donner ses arguments. ce qui donne

```
procedure reduction (var num, den : integer);
    begin
        n :=num ;
        d :=den ;
        while d<>0 do
            begin
            reste :=n mod d ;
            n :=d ;
            d :=reste ;
            end ;
        if n>1 then
            begin
            num :=num div n ;
            den :=den div n
            end
    end ;
```

Pour terminer notre programme, nous avons besoin d'un procédure qui additionne deux nombres rationnels en les réduisant au même dénominateur.

```
procedure somme (var n,d : integer ; n1,d1,n2,d2 : integer) ;
    begin
    n :=n1*d2+n2*d1 ;
    d :=d1*d2
    end ;
```

Nous pouvons maintenant donner le programme complet qui calcule les différentes sommes partielles et les affiche à l'écran.

```
program sommeharmonique (input,output) ;
const
    premierterme  = 2 ;
var
    numerateur,denominateur,dernierterme,termecourant : integer ;

procedure reduction (var num, den : integer);
    var
        n,d,reste : integer ;
    begin
        n :=num ;
        d :=den ;
```

```
            while d<>0 do
                    begin
                    reste :=n mod d ;
                    n :=d ;
                    d :=reste ;
                    end ;
            if n>1 then
                    begin
                    num :=num div n ;
                    den :=den div n
                    end
    end ;

procedure somme (var n,d : integer ; n1,d1,n2,d2 : integer) ;
    begin
    n :=n1*d2+n2*d1 ;
    d :=d1*d2
    end ;

begin
    numerateur := 1 ;
    denominateur := 1 ;
    write ('Combien de terme calculer :    ') ;
    readln (dernierterme) ;
    for  termecourant := premierterme to dernierterme  do
            begin
            somme (numerateur, denominateur, numerateur, denominateur, 1,
                                                                    termecourant) ;
            reduction (numerateur, denominateur) ;
            writeln(numerateur :1,' / ',denominateur :1)
            end
end.
```

Variables et paramètres.

Nous avons parlé de variables et de paramètres. Quelles sont les différences entre **variables locales**, **variables globales** et **paramètres** ? Examinons la distinction sur un exemple.

```
program program_12 (input,output) ;
var
    x : integer ;
procedure change ;
    begin
    x :=1
    end ;

begin
    x :=0 ;
    change ;
    write(x)
end.
```

Dans ce programme la variable x est une **variable globale**. Si l'on exécute ce programme, la variable x est initialisée à 0 et lors de l'appel de la procédure *change* sa valeur est modifiée. regardons le programme suivant

```
program program_13 (input,output) ;
var
    x : integer ;
procedure change ;
    var
        x : integer ;
    begin
    x :=1
    end ;
begin
    x :=0 ;
    change ;
    write(x)
end.
```

Il y a deux variables *x* l'une est une variable globale et l'autre est une **variable locale** à la procédure *change*. Ce programme affichera la valeur 0. Donnons maintenant un paramètre à la procédure *change.*

```
program program_14 (input,output) ;
var
    x : integer ;
procedure change (var y : integer);
    begin
    y :=1
    end ;
begin
    x :=0 ;
    change (x);
    write(x)
end.
```

Ce programme affichera la valeur 1. Nous avons effectué un **passage de paramètre par variable**, la variable *y* est « synonyme » de *x*. Les modifications faites à la variable *y* le sont aussi à la variable *x*. Avec ce genre de passage de paramètre, le paramètre doit être un variable on ne pourra pas avoir ;

```
…
change (5) ;
change(3*x) ;
```

La deuxième manière de passer un paramètre est de la faire sans le mot clé *var*. Dans ce cas il s'agit d'un **passage de paramètre par valeur**. Comme son nom l'indique, seule la valeur est attribuée à la variable.

```
program program_15 (input,output) ;
var
    x : integer ;
procedure change (y : integer);
    begin
    y :=1
    end ;

begin
    x :=0 ;
    change (x);
    write(x)
end.
```

Ce programme affiche la valeur 0. En effet, lors de l'appel de la procédure la variable *y* prend la valeur 0, elle est ensuite modifiée mais pas la variable x. Les appels de procédures suivants sont acceptés :

```
…
change (5) ;
change(3*x) ;
```

FONCTIONS.

Une fonction, comme son nom l'indique, est une opération sur un ou plusieurs arguments qui renvoie une valeur de type réel, entier ou autre type simple.

Nous connaissons déjà les fonctions prédéfinies telles que *sqrt*, *sqr*, *abs* … Le plus c'est que nous pouvons définir nos propres fonctions comme nous l'avons fait pour les procédures. A la différence qu'à une procédure était associée une action et à une fonction est associée une valeur ; l'image de la fonction.

Syntaxiquement, un appel de procédure peut-être considéré comme une instruction tandis qu'un appel de fonction est considéré comme un facteur. En utilisant les fonction *sqrt* et *sqr*, on peut écrire pour rechercher l'hypothénuse d'un triangle;

```
program programme_triangle (input, output) ;
var
    c1,c2 : real ;
    h : real ;
begin
    write (‘Entrer les deux cotés du triangle : ‘) ;
    readln(c1,c2) ;
    h :=sqrt(sqr(c1)+sqr(c2)) ;
    write(‘Hypothénuse : ‘,h)
end.
```

Dans ce programme on pourrait remplacer l’utilisation de la fonction *sqr* par une fonction basée sur l’algorithme du chapitre précédent. Une déclaration de fonction étant semblable à une déclaration de procédure, à ceci près qu’il faut préciser le type de la variable que la fonction renvoie, on obtient;

```
program programme_triangle2 (input, output) ;
var
    c1,c2 : real ;
    h : real ;

function racinecarree( x:real) : real;
const
    epsilon = 1E-6;
var
    racine : real;
begin
    if x=0 then
            racine := 0
            else
            begin
            racine := 1;
            repeat
                racine:=(x/racine+racine)/2
            until abs(x/sqr(racine)-1)<epsilon ;
            end;
    racinecarree := racine
end ;

(*Corps du programme*)
begin
    write (‘Entrer les deux cotés du triangle : ‘) ;
    readln(c1,c2) ;
    h :=racinecarree(sqr(c1)+sqr(c2)) ;
    write(‘Hypothénuse : ‘,h)
end.
```

Nous avons vu dans les définitions de procédures et de fonctions que l'on pouvait déclarer des variables locales, des variables globales, des paramètres passés à une procédure ou à une fonction … Il faut être prudent lorsque l'on rédige un programme lors du choix de ces variables. En effet, si malencontreusement, vous modifiez une variable globale au sein d'une procédure ou si vous confondez un passage de paramètres par valeurs et par variables, votre programme risque de ne pas avoir l'effet escompté ! On parle dans ce cas d'**effet de bord**. Dans certains cas, le programmeur peut générer des effets de bord dans son programme pour apporter un plus dans la rapidité d'exécution. Attention cependant, car ce genre d'approche nuit généralement à la lisibilité du programme et en programmation, la clarté dans l'écriture est primordiale. De plus, lorsque vous écrivez des fonctions (comme en mathématique) il est logique que celle-ci ne tiennent compte que des paramètres qui lui sont donnés.

Exercices.

Ecrire une fonction *chiffre(n,k)* qui retourne la valeur du $k^{ième}$ chiffre depuis la droite du nombre *n*. Par exemple : *chiffre (254693,2)=9*

5 Structure des données.

TYPES SIMPLES PREDEFINIS

Nous avons déjà rencontré quelques types simples du Pascal, à savoir, les entiers, les réels et les chaines de caractères. Donnons-en maintenant une liste plus complète ; **byte, integer,word, longint, shortint, real, boolean, ..**

TYPES SIMPLES

L'utilisateur peut lui aussi définir ses propres types de variables. Il a pour ce faire les types **énuméré**, **intervalle** et **ensemble**.

Le *type énuméré* sert, comme son nom l'indique, à définir un ensemble de valeurs particulières sur laquelle est définie une notion d'ordre. Cette notion est utile lorsqu'une variable ne peut prendre qu'un petit nombre de valeurs. L'utilisation d'un type énuméré ajoute à la lisibilité du programme.

La définition d'une variable doit être précédée de la déclaration de son type. Par exemple ;

```
type
    TJourSemaine = (Dimanche, Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi)
    TGlace = (Vanille, Chocolat, Fraise)
    TNotes = (Do, Ré, Mi, Fa, Sol, La, Si, Do)
var
    Note : Tnotes
    J : TJourSemaine
```

Le *type intervalle* est fort semblable au type énuméré si ce n'est qu'il se défini comme un sous-ensemble d'un ensemble connu (les entiers, les caractères, un type énuméré). Attention cependant, un type intervalle ne peut pas contenir de trou. Quelques exemples d'intervalles;

```
type
    Tjours = 1..31
    TjoursOuvrables = Lundi ..Vendredi
    Tchiffre = '0'..'9'
```

La définition du type TjoursOuvrables ne peut se faire que si le type TjourSemaine est défini.

Une notion d'ordre est définie sur les type énuméré et intervalle, on peut appliquer les opérateurs **ord**, **pred** et **succ** aux éléments d'un type énuméré ou intervalle. Par exemple

```
ord(Lundi)=1
pred(Chocolat)=Vanille
```

Un *ensemble* est une collection d'objets de même type. Si *S* est un ensemble d'objets de type *T* , alors tout objet de type *T* est, ou n'est pas, un élément de l'ensemble. On peut définir un type **set** en correspondance avec n'importe quel type énuméré. Par exemple ;

```
type
    ingredients =  (pommes, fraises, bananes, noisettes, creme, creme-glacee, creme-
                    chocolat,  sucre, glace) ;
    dessert = set of ingredients ;
var
    sorbets, salade-de-fruits : dessert
```

Le type *dessert* est le type ensemble associé au type énuméré *ingrédients*. Les variables *sorbets* et *salade-de-fruits* peuvent prendre les valeurs [creme-glacee,creme-chocolat] ou [sucre] ou encore l'ensemble vide [].

Les opérations ensmblistes union (+), différence (-), intersection (*), contient (≥), est contenu par et appartient (**in**) sont disponibles.

Exemple : Jeu de l'oie

Nous allons simuler un jeu de l'oie. Supposons que ce jeu soit le suivant. Il faut déplacer son pion des cases 1 à 63. Celui, qui du premier coup, jette 6 et 3, se place sur la case 26, et celui qui jette 4 et 5 se place sur la case 53. Le joureur qui tombe sur une oie avance une seconde fois du même nombre de cases. Celui qui s'arrête sur le pont (case 6) avance à la case 12. Le joueur qui arrive à l'hotel (case 12) passe deux tours. Celui qui tombe dans le puit (case 31) y reste jusqu'à ce qu'un autre joueur vienne le délivrer. Celui qui arrive au labyrinthe (case 42) retourne à la case 30. Le joueur qui arrive en prison (case 52) attend sa délivrance. Celui qui meurt (case 58) retourne à la case départ (case 1). Celui qui est rejoint par un autre va se mettre à la case d'où il vient. Le joueur qui dépasse 63 recule d'autant de cases supplémentaires. Celui qui atteint exactement le numéro 63 a gagné.

Nous allons écrire un programme qui simule le mouvement d'un joueur afin de présenter les régles au fur et à mesure du jeu.

L'algorithme sera de le suivant

```
initialisations
répéter
        entrer les valeurs des dés
        calcul de la position
        si non gagné alors
                si position =9 alors
                        vous êtes sur la première oie
                si position est une autre oie alors
                        avancer encore
                si position > 63 alors
                        reculer
                        si une oie alors reculer encore
                vous êtes sur une case : est-elle banale ou gage ?
jusqu'à gagné
```

On obtient ;

```
program jeudeloie (input,output) ;
const
    dernierecase = 63 ;
type
    Tcase = 1..dernierecase ;
    enscase = set of Tcase ;
    Tpos = 0..dernierecase ;
var
    surprises : enscase ;
    position : Tpos ;
    de1,de2,points,calculpos : integer
    gagne : boolean ;

procedure lancer(var de1,de2 : integer) ;
var
    correct : bolean ;
begin
    writeln('Rentrez la valeur des dés : ') ;
    repeat
        read(de1,de2) ;
        correct := (de1 in [1..6]) and  (de2 in [1..6]) ;
        if not correct then
            writeln('Les dés ont six faces ! ') ;
    until correct ;
end ;

begin
    surprises := [6,19,31,42,52,58] ;
    position :=0 ;
    gagne := false ;
```

```
    repeat
        writeln('Vous êtes sur la case  ',position :3) ;
        writeln('Regardez s''il y a un pion') ;
        lancer (de1,de2);
        points :=de1+de2 ;
        calculpos :=position + points ;
        gagne := calculpos = dernierecase ;
        if not gagne then
            begin
            if calculpos = 9 then
                if  (de1=6) or (de2=6) then
                    calculpos := 26
                else
                    calculpos := 53 ;
            if (calculpos mod 9 =0) and (calculpos <>9) then
                calculpos := calculpos + points ;
            if calculpos > dernierecase then
                begin
                position := dernierecase – calculpos mod 63 ;
                if position mod  9 =0 then
                    position := position – points
                end
            else
                position := calculpos ;
            if position in surprise then
                case position of
                    6       : position := 12 ;
                    12      :writeln('Passez deux tours ') ;
                    31, 52  :writeln('Patience !') ;
                    42      :position := 30 ;
                    58      :position :=1 ;
                end
                else
                    writeln('Case bien banale.') ;
    until gagne ;
    writeln('Vous avez terminé … et gagné si vous êtes le premier !') ;
end.
```

TYPES STRUCTURES : VECTEURS ET TABLEAUX

Les variables de type structurés diffèrent des variables de type simple en ce sens qu 'elles se composent de plusieurs composant de type simple.

Un **vecteur** est un ensemble de données toutes du même type simple et ayant le même nom, les différentes données se distinguent par un indice. On pourra définir un vecteur de 10 entiers comme suit ;

```
vecteur : array [0..9] of integer ;
```

On accède à chaque variable par son indice, voici quelques exemples ;

```
vecteur[5] := 123 ;

for i :=0 to 9 do
    vecteur[i] :=i*10 ;

for i :=2 to 9 do
    vecteur[i] :=vecteur[i-1]+vecteur[i-2] ;
```

L'utilité d'un vecteur se fait sentir lorsque l'on est face à un, relativement, grand nombre de variables de même type et ayant le même rôle. Si l'on veut écrire un programme qui résoud une équation du second degré, on choisira pour représenter les coefficients du polynôme les variables *a*, *b* et *c*. Si maintenant on veut écrire un programme qui résoud une équation de degré quelconque n, il est clair que ce même genre de choix n'est plus d'actualité. D'une part, l'écriture est lourde et d'autre part, que fait-on si n>26 ? On préfèrera définir un vecteur a de taille n comme suit

```
a : array [1..n] of real ;
```

En PASCAL, ce genre de déclaration ne peut se faire que si *n* est connu c'est-à-dire déclaré comme étant une constante.

Le lecteur attentif se demande peut-être s'il est possible de représenter des matrices en PASCAL. Le type **tableau** est un vecteur à plusieurs dimensions, par exemple,

```
type
    TMatrice : array [1..3] of array [1..3] of integer ;
var
    M1,M2 : TMatrice ;
```

M1 et M2 sont définies comme matrices 3×3 et l'on pourrait définir des opérations sur ces matrices carrées. Les deux écritures suivantes sont acceptées pour faire référence à un élément du tableau (si l'on suppose *i* et *j* définis).

```
M1[i][j]    ou    M1[i,j]
```

La valeur d'un tableau peut être affectée à un autre tableau de même type par une affectation unique ; M1 :=M2. Un tableau peut être passé en paramètre à une fonction, on pourrait donc écrire un programme qui calcule le déterminant d'une matrice carrée.

```
function determinant (M : TMatrice) :integer ;
begin
    determinant :=      M[1,1] * (M[2,2]*M[3,3] - M[2,3]*M[3,2]) –
                        M[2,1] * (M[1,2]*M[3,3] - M[3,2]*M[1,3]) +
                        M[3,1] * (M[1,2]*M[2,3] - M[2,2]*M[1,3]) ;
end ;
```

Par contre si nous voulons écrire une procédure qui calcule la somme de deux matrices, nous ne pouvons pas utiliser une fonction car la somme de deux matrices est aussi une matrice et une fonction retourne un type simple. Mais nous pouvons écire une procédure ;

```
procedure somme (M1,M2 : TMatrice ; var M : TMatrice) ;
var
    i,j : integer ;
begin
    for i :=1 to 3 do
        for j :=1 to 3 do
            M[i,j]= M1[i,j]+ M2[i,j] ;
end ;
```

Le problème de cette manière de faire est que la matrice M est recopiée en mémoire, ce qui risque de prendre de la place si l'on écrit ce genre d'instructions pour des grands tableaux.

Un vecteur particulier en PASCAL est le type **string** qui est un vecteur de caractères. Quelles sont les différences entre les deux déclarations suivantes ;

```
type
    Tt1 = array [0..255] of char ;
    Tt2 = string ;
var
    t1 : Tt1 ;
    t2 : Tt2 ;
```

Pour le type Tt1, PASCAL ne tient pas compte des informations déjà stockées dans la variable t1. C'est un vecteur contenant des caractères et ce vecteur à une taille de 256. Si je veux l'écrire, il faut que je fasse une boucle.

Tandis que pour le type Tt2, le premier caractère (t2[0]) contient la longueur du string. Si vous faites l'affectation suivante ;

```
t2 := 'Jean'
```

t2[0] contient le caractère 4, t2[1] contient 'J', t2[2] contient 'e' … Le type string permet des opérations supplémentaires à celles autorisées sur un vecteur. Vous pouvez comparer ou concaténer deux strings, vous pouvez affecter une valeur à un string via un readln par exemple.

TYPES STRUCTURES : ENREGISTREMENTS

Tout comme un vecteur, un enregistrement se compose de plusieurs variables. Pour un enregistrement, les variables peuvent être de type différent et elles se différencient par un nom et plus par un indice. Par exemple, nous pourrions vouloir définir un type personne pour pouvoir reconnaître nos amis. On répertorierait un ami en lui donnant un prénom, un nom, un age et une valeur boléenne ami ou pas. On définit le type

```
type
    TPersonne = record
            prenom, nom : string[30] ;
            age : integer ;
            ami : boolean ;
    end ;
var
    jean : TPersonne ;
```

Les affectations se font comme suit ;

```
jean.prenom := 'Jean' ;
jean.age := 21 ;
```

Les combinaisons de types sont évidemment possibles, on peut définir un groupe de voisins, de collègues …

```
var
    voisins : array [1..10] of TPersonne ;
    copins-de-classe : array [1..10]  of TPersonne ;
```

On peut alors exécuter les instructions suivantes ;

```
for i :=1 to 10 do
    voisins[i].ami := false ;
voisins[1].prenom := 'Jean' ;
for i :=1 to 10 do
    copins-de-classe[i].age := 19 ;
```

On peut affecter directement la valeur de tous les champs d'un enregistrement à un enregistrement du même type.

```
voisins[9] := jean ;
```

Exercices

1. Calculer la somme et la moyenne des éléments d'un vecteur de *n* entiers donnés.
2. Rechercher le minimum et le maximum dans un tableau d'entiers donnés.
3. Soit un vecteur de n entiers, on demande de « renverser » ce vecteur; les premiers seront les derniers. NB ; on n'utilisera pas d'autre vecteur que celui donné au départ.
4. Soit un vecteur de n entiers V, on demande de faire tourner les éléments du vecteur jusqu'à ce qu'un élément donné VAL de ce vecteur soit en tête.
5. Ecrire un programme qui résoud un système de deux équations à deux inconnues par la méthode de Gauss.
6. Soit un tableau de N×P×Q entiers, placer les éléments dans l'ordre dans un vecteur.
7. Soit un tableau ECHEC, simulant un jeu d'échec (8×8) sur lequel se trouvent deux reines. Toutes les cases sont à 0 sauf celles où se trouvent les reines (1). Ecrire un programme qui précise si les reines peuvent se prendre.
8. Ecrire un programme qui inverse un string (essayer avec un palindrome).

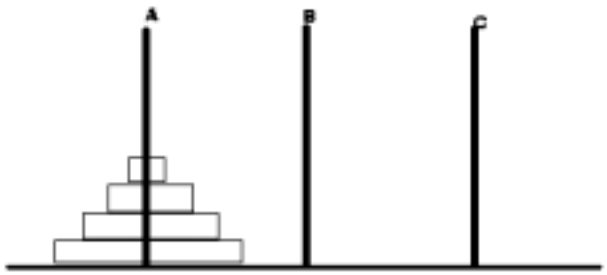
6 Récurcivité.

Nous avons vu plus tot les notions de procédures et de fonctions. La notion de récurcivité est très importante en informatique. Elle repose sur le principe suivant ; supposons que l'on veuille calculer la factorielle d'un nombre *n* il sufit en fait de multiplier *n* à la factorielle de *n-1* . Si nous déclarons une fonction *fact(n)* nous pouvons la défnir comme suit ;

fact (n) = n * fact (n-1) ;

Et nous obtenons une procédure qui s'appelle elle-même. PASCAL permet ce genre d'appel de fonction. On parlera alors d'appel **récurcif**.

Vers la fin du dix-neuvième siècle est apparu un jeu appelé les tours de Hanoï. Ce jeu est constitué de 3 batons en bois, sur l'un deux est placé un certain nombre de disques de diamètre décroissant. Le but du jeu est de déplacer sur un des deux autres batons toute la pile de disques sans jamais poser un disque sur un disque de diamètre inférieur et en ne déplacant qu'un seul disque à la fois. L'histoire raconte que les bonzes du temple de Brahma étaient détenteurs de 64 disques et que la fin de leur jeu signifierait la fin du monde.



Supposons que notre jeu ne se compose que de 4 disques. L'opération que nous voulons effectuer peut se noter ; *déplacer (4,A,B,C)* ce qui signifiera déplacer 4 disques du baton A vers le baton B en passant par C. Si nous supposons que l'on sait comment déplacer les trois disques supérieurs, la solution serait ;

déplacer (3,A,C,B)
déplacer le dique de A vers B
déplacer (3,C,B,A)

En raisonnent comme cela, nous avons résolu notre problème car les étapes suivantes seront ;

déplacer (2,A,B,C)
déplacer le dique de A vers C
déplacer (2,B,C,A)

Il suffira de préciser qu'il ne faut rien faire lorsque le nombre de disques est nul. Et nous obtenons une procédure récurcive, i.e. qui s'appelle elle-même.

```
program Hanoï;
type
   Tbaton =(G,C,D);

const
    libelle : array [0..2] of string = ('Gauche','Centre','Droite');

var
  Borigine,Bdestination,Bintermediaire : baton;
  n : integer;

procedure Hanoi(n : integer;Borigine,Bdestination,Bintermediaire:baton);
begin
       if n=1 then
          Writeln('D,placer ',libelle[ord(Borigine)], ' vers ',libelle[ord(Bdestination)])
       else
          begin
          Hanoi(n-1,Borigine,Bintermediaire,Bdestination);
          Writeln('D,placer ',libelle[ord(Borigine)], ' vers ',libelle[ord(Bdestination)]);
          Hanoi(n-1,Bintermediaire,Bdestination,Borigine)
          end;
end;

begin
    write('Combien de batons ?  ');
    readln(n);
    Hanoi(n,G,C,D);
    readln
end.
```

Attention cependant lors de l'utilisation de programmes récurcifs, ils coûtent chers en espace mémoire . En effet, à chaque appel récurcif de la procédure, les variables passées en paramètres sont recopiées, un emplacement mémoire est à nouveau alloué aux variables. Pour vous en convaincre, testez ce programme avec des valeurs de plus en plus grandes du nombre de disques à déplacer (5 ; 6 …).

Lorsque l'on écrit une procédure récurcive, i.e. qui s'appelle elle-même il vaut mieux être sur que ces appels récurcifs ne seront pas infinis. Pour ce faire il est préférable d'écrie une procédure récurcive sous la forme ;

```
procedure <nom-procedure-récurcive> (<paramètres>) ;
    if <cas trivial> then
            solutions du cas trivial (plus d'appel de la procédure)
    else
            <nom-procédure-récurcive> (<paramètres>) ;
end ;
```

Exercices

Ecrire un programme qui permet de calculer les nombres de Fibonnacci. Ecrire un programme récurcif et un programme itératif. Préciser ensuite pourquoi le choix d'une procédure récurcive n'est pas judicieux dans ce cas.

$F_1=1$, $F_2=1$, $F_n=F_{n-1}+F_{n-2}$

7 Fichiers.

8 Structures de données dynamiques : les pointeurs.

Chapitre 4 ASSEMBLEUR, LANGAGE D'ASSEMBLAGE.

Chapitre 5 EXCEL, LANGAGE NIVEAU UTILISATEUR.