

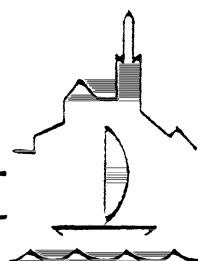
ELEMENTS DE PROGRAMMATION

par

HERVÉ LEHNING

Atelier Logiciel
du
Lycée Janson de Sailly

ALESUP



Publication de l'Institut P. S. M. à Aix-Marseille

7

1987

ISSN 0297-4347

Copyright 1987, IREM D'Aix-Marseille

ELEMENTS DE PROGRAMMATION

A.L.E. Sup.

```
*****  
*                                     *  
*   ELEMENTS DE PROGRAMMATION   *  
*                                     *  
*****
```

Version 1.1 du 04-2-86

par Hervé Lehning

Atelier Logiciel Janson de Saily

Ce cours de programmation élémentaire est destiné aux scientifiques désireux d'apprendre à réaliser de petits programmes de calcul numérique. Il est issu d'un cours d'initiation à la programmation donné aux élèves de Terminales C et de Classes Préparatoires au lycée Janson de Saily.

Il suppose que le lecteur est déjà familiarisé au maniement des ordinateurs (compatibles IBM PC) et à l'utilisation des logiciels les plus courants (traitement de textes, tableur, traceur de courbes).

Il utilise le langage de programmation Pascal dans sa version Turbo.

Siège social de l'A.L.E. Sup :

C.I.R.M. 70 route Léon Lachamp 13009 Marseille

Adresse de l'Atelier Logiciel Janson de Saily :

106 rue de la Pompe 75016 Paris

Introduction

Il est difficile de présenter le langage Pascal progressivement car la programmation dans ce langage exige une bonne compréhension des structures de données utilisées ainsi que de la structure des programmes elle-même en modules et procédures.

Cependant, dans ce polycopié, je vais tenter d'en proposer une introduction progressive en limitant volontairement les structures de données. De plus, certaines possibilités ne sont pas abordées. Le lecteur pourra compléter sa formation directement dans l'ouvrage de Niklaus Wirth : "Introduction à la programmation systématique" (Masson) s'il le désire. Cependant la plupart des programmes de calcul numérique sont accessibles avec les notions exposées dans ce polycopié.

Le langage Pascal, même s'il a d'énormes avantages par rapport aux langages non structurés (comme les versions les plus courantes du Basic par exemple), a tout de même de gros défauts. En particulier, il ne prévoit pas l'utilisation de composants logiciels (parties de programmes préécrites pour accomplir certaines tâches) et nous devons les utiliser de manière bien inélégante! En effet, nous sommes obligés de les considérer comme des procédures internes au programme et donc de les insérer dans la liste de celui-ci. En particulier, ces composants ne peuvent être compilés séparément.

Un dernier conseil : prenez le temps de faire les exercices des premiers chapitres car ils vous habitueront à la syntaxe du Pascal (il est si facile d'oublier un point-virgule, ou de confondre ":", "=", "et" et "!") et également à l'éditeur du Turbo ce qui n'est pas une sinécure!

Hervé Lehning

Table des matières

Introduction	page 2
Table des matières	page 3
Rudiments de programmation	
Analyse d'un problème	page 4
Les objets manipulés	page 5
Un exemple simpliste :	
le calcul de l'aire d'un disque	page 7
Structure d'un programme	page 7
Utilisation du compilateur	page 8
Exercices	page 8
Les structures de contrôle de base	
Instruction if ... then ... else	page 9
Un exemple :	
résolution d'une équation du second degré	page 10
Les boucles	page 11
Invariant de boucle	page 12
Exercices	page 12
Les fonctions et les procédures	
Les fonctions	page 14
Les procédures	page 17
Utilisation d'une bibliothèque	page 18
Un exemple :	
résolution d'une équation par dichotomie	page 19
Exercices	page 21
Les tableaux	
Les tableaux	page 23
Problème de la dimension	page 26
Exercices	page 27
Bibliothèque Modulo	
Présentation générale	page 28
Index	page 29

Chapitre I

Rudiments de programmation

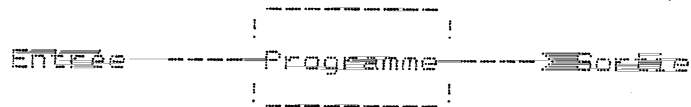
Le but de ce premier chapitre est de vous permettre une première familiarisation avec la programmation en Pascal et avec les commandes du Turbo. Les exemples utilisés n'ont donc aucun intérêt en eux-mêmes, leur seul but étant l'apprentissage de ces rudiments.

1. Analyse d'un problème.

La familiarisation à l'utilisation de logiciels a pu vous montrer que, vu de l'extérieur, l'exécution d'un programme transforme un ordinateur en une espèce de "boîte noire" :

pour un certain nombre d'entrées, celle-ci fournit en réponse un certain nombre de sorties.

Pour le mathématicien, il s'agit de l'évaluateur d'une certaine fonction.



Le programmeur sait quelles sorties il désire obtenir, il lui faut définir les entrées et surtout l'algorithme (c'est-à-dire la suite des opérations à effectuer transformant ces entrées en les sorties désirées).

Cette première étape du travail s'appelle l'analyse du problème (ce qui n'est d'ailleurs pas propre à l'informatique mais encore plus essentiel dans son cas). Les premiers exemples que nous rencontrerons sont trop simples pour mettre cette partie en valeur mais il faut bien noter qu'elle est essentielle comme le montre les derniers exemples.

Elle comprend :

* la formulation correcte du problème.

Ce qui n'est pas si simple car le mot correct doit être pris dans un sens assez éloigné du sens commun qui, sans doute, ne va se dégager parfaitement pour vous que progressivement tout au long de votre initiation. Pour l'instant, notez que vous devez être sûr qu'un automate seulement capable d'effectuer un certain nombre d'opérations élémentaires sur des objets de

types bien définis puisse résoudre le problème en suivant votre algorithme pas à pas et ceci sans nouvelle intervention de votre part que votre description préalable.

* la recherche d'un algorithme de résolution.

avec en général une décomposition en sous algorithmes (qui correspondront à autant de modules ou de procédures Pascal).

* la vérification logique de l'algorithme.

c'est-à-dire : l'algorithme fournit-il bien le résultat escompté en un temps fini (et raisonnable) ?

Après ces trois étapes vient seulement la rédaction du programme dans le langage de programmation utilisé puis ... la mise au point de ce programme. Le temps consacré à cette dernière tâche sera d'autant plus court que les étapes précédentes auront été bien travaillées.

2. Les objets manipulés.

Pour discuter davantage de l'analyse d'un problème, il faut préciser sur quels objets travaille l'ordinateur et quelles opérations il est capable de leur faire subir.

A. Variables.

Pour programmer un ordinateur, il n'est pas nécessaire (et même sans aucun intérêt) de connaître ces objets physiquement : il suffit de penser qu'il s'agit de "cases mémoires" que nous appelons variables.

Le programmeur donne un nom à chacune des variables qu'il entend utiliser, nom que nous appelons identificateur de la variable : il est conseillé d'éviter des identificateurs tels que AHTDBVHT ou CHTRONG ce qui manque de clarté, il vaut mieux que l'identificateur ait un sens pour un lecteur humain (bien entendu, il n'a aucun sens de ce type pour le compilateur). Par exemple, une variable destinée à contenir la valeur numérique du rayon d'un cercle peut s'appeler rayon ou rayon_du_cercle.

La syntaxe d'un identificateur est simple : il ne doit pas commencer par un chiffre, ne pas contenir de blanc, ne doit pas être un des mots réservés du langage et ne pas dépasser une ligne (c'est-à-dire 127 caractères).

Ces variables ne peuvent être totalement quelconques mais doivent appartenir à un type soit prédéfini dans le langage Turbo Pascal, soit que l'on définit en début de programme.

B. Types de variables.

En Pascal, les types fondamentaux indispensables en calcul scientifique sont :

- le type entier (Integer)
en fait seulement de -32767 à 32767
- le type réel (Real)
c'est-à-dire les nombres avec virgule.
Plus précisément à 1 chiffre significatif sur l'échelle de $1 \text{ E } -38$ à $1 \text{ E } 38$, l'utilisation du coprocesseur 8087 porte le nombre de chiffres à 16 et l'échelle de $4.19 \text{ E } -307$ à $1.6 \text{ E } 308$.
- le type booléen (Boolean)
qui n'a que deux valeurs possibles : vrai ou faux
(True ou False), bien utile pour les tests.

Ces types sont dits types scalaires par opposition avec d'autres types que nous verrons ultérieurement.

En début d'un programme, nous devons déclarer les variables et leurs types de la façon suivante :

```
(Identificateur), (Identificateur), ... : (type);  
Par exemple :  
Var    pi, rayon_du_cercle, aire_du_cercle : Real;
```

C. Opérations licites.

Chaque type de variables définit un champ sur lequel un certain nombre d'opérations sont licites.

Parmi celles-ci, l'affectation d'une valeur à une variable (c'est-à-dire le stockage dans la case mémoire en question de cette valeur) est toujours licite.

Sa syntaxe est :

```
(Identificateur) := (expression);
```

Où l'expression doit être du même type que la variable, le terme expression a le même sens qu'en Mathématiques.

Par exemple :

```
pi := 3.14159;  
aire_du_cercle := pi * rayon_du_cercle * rayon_du_cercle;
```

Les autres opérations licites sont les opérations usuelles en Mathématiques, nous ne nous étendrons donc pas.

D. Constantes.

Certaines variables sont en fait destinées à contenir des constantes, il est alors préférable de les déclarer comme constantes. Les constantes sont déclarées avant les variables. En pascal, leur type est attribué de façon implicite (cette facilité peut être bien ennuyeuse : si on désire que l soit un réel, il faut l'écrire 1.0).

Sa syntaxe est :

```
(Identificateur) = (valeur);
```

Par exemple :

```
Const    pi = 3.14159;
```

3. Un exemple simpliste : le calcul de l'aire d'un disque.

Dans ce cas, le problème est clair : il s'agit d'obtenir l'aire d'un disque en sortie, l'entrée étant son rayon, voici la liste du programme :

Program Disque ;

```
Const    pi = 3.14159 ;
Var      rayon, aire : Real ;
```

Begin

```
  clrscr ; (* nettoie l'écran *)
  writeln('donnez le rayon du disque en mètres') ;
  readln(rayon) ;
  writeln ;
  aire := pi * rayon * rayon ;
  write('l'aire est égale à ', aire, ' mètres carrés.')
```

End.

Nous allons utiliser cet exemple pour donner la structure d'un programme.

4. Structure d'un programme.

Comme vous pouvez le remarquer, un programme écrit en Pascal commence toujours par un en-tête qui est Program suivi du nom du programme choisi comme il l'entend par le programmeur, suivi d'un point-virgule. Notez dès à présent que toutes les instructions doivent se terminer par un point-virgule.

Le paragraphe suivant est le paragraphe de déclaration des constantes toujours précédé de l'en-tête Const, il est suivi ici du paragraphe de déclaration des variables toujours précédé de l'en-tête Var. De façon générale, il est précédé par le paragraphe de déclaration des types mais nous verrons cette question plus tard.

Le paragraphe des instructions commence par l'en-tête Begin et finit par End. (toujours un point et un retour-chariot à la fin).

Vous pouvez remarquer qu'il est indenté, que des lignes blanches ont été ménagées pour distinguer les groupes d'instructions : ceci n'est pas obligatoire pour le compilateur (et en fait celui-ci l'ignore) mais aide à la compréhension du programme par un lecteur humain. Pour la même raison, des remarques sont mises entre (* et *). Ceci n'est également pas pris en compte par le compilateur.

Nous rencontrons ici les procédures d'entrée au clavier et de sortie à l'écran. Vous pouvez à cette occasion remarquer quelle est la syntaxe de l'appel à une procédure

on écrit l'identificateur de la procédure (ici `clrscr`, `writeln` ou `readln`) suivi de la liste de ses paramètres (on précisera cette notion plus tard) entre parenthèses et séparés par des virgules. Lors de l'exécution, le contrôle de l'ordinateur est donné à la procédure en question.

5. Utilisation du compilateur.

Pour rendre ce programme exécutable, il est nécessaire de l'écrire sur l'éditeur du Turbo-Pascal puis ensuite de le compiler (c'est-à-dire transformer son écriture en langage pascal en une écriture en langage machine qui seul est exécutable). Pour cela, on charge le "Turbo" (ou "Turbo-67" pour utiliser le coprocesseur) sous le DOS. Quand cette opération a été accomplie, le Turbo demande si l doit inclure les messages d'erreurs, répondez "Y" pour oui. Ensuite apparaît le menu :

Edit	Compile	Run	Save
Dir	Quit	compiler Options	

On choisit l'option désirée en tapant la majuscule correspondante, ici nous tapons donc "E".

L'éditeur du Turbo s'utilise comme un traitement de textes usuel. Pour en sortir et revenir au menu, la commande usuelle est "ctrl-K ctrl-D" (mais il est possible d'en changer grâce au programme d'installation du Turbo TINST ou d'éventuelles surprises).

Une fois fini de taper le texte, nous sortons donc de l'éditeur puis quand le signe > apparaît nous tapons "C" pour le compiler. La compilation est alors très rapide c'est pour cela que le langage s'appelle Turbo! Puis nous tapons "R" pour exécuter le programme.

Pour lister le programme sur imprimante, le plus simple est d'utiliser la fonction Print du DOS.

6. Exercices.

Les exercices qui suivent sont destinés à donner un minimum d'expérience d'écriture d'un programme et d'utilisation de l'éditeur ainsi que du compilateur Turbo-Pascal.

1. Ecrivez le programme précédent sur l'éditeur du Turbo, compilez-le et exécutez-le.
2. Ecrivez un programme de calcul d'une expression du type $ax^2 + bx + c$.
3. Ecrivez un programme de calcul de la somme de deux nombres. De même avec le produit.

Structures de contrôle de base

Jusqu'à présent, nous ne sommes en mesure que de faire des programmes linéaires (c'est-à-dire que chaque instruction est exécutée une seule fois et toutes les instructions dans l'ordre où elles ont été écrites). Ainsi nous ne pouvons même pas programmer la résolution d'une équation du second degré. Les instructions conditionnelles permettent de prévoir les différents cas de figures dans un calcul, dans ce chapitre nous voyons la plus simple d'entre elles : l'instruction `if ... then ...`.

De même, il est rare qu'un algorithme de résolution ne nécessitent pas la répétition d'un certain nombre de calcul tant qu'une certaine condition est réalisée (où jusqu'à ce que ...), ce qui en toute logique revient au même : ce sont les instructions `while ... do ...` et `repeat ... until ...`.

1. L'instruction `if ... then ... else ...`

La syntaxe de cette instruction est :

```
if (expression booléenne) then (instruction)
    [ else (instruction) ] ;
```

Où la dernière partie a été mise entre crochets car elle peut être omise. Remarquez qu'il ne doit pas y avoir de point-virgule avant le mot `else`.

Une expression booléenne est une expression dont le résultat est une valeur booléenne, elle se construit avec les opérateurs "`>`", "`<`", "`>=`", "`<=`", "`=`", "`<>`" et "`and`", "`or`" et "`xor`".

Lors de l'exécution de cette instruction, l'expression booléenne est évaluée; si cette valeur est vraie alors l'instruction suivant le "`then`" sera exécutée sinon ce sera l'instruction suivant le "`else`". Si la valeur est fausse et le "`else`" absent, c'est l'instruction suivante du programme qui sera exécutée.

Une instruction peut être composée de plusieurs instructions, dans ce cas on l'écrit :

```
begin
    (instruction 1) ;
    (instruction 2) ;
    .....
    (instruction n) ;
end
```

ce qui constitue une instruction composée, cette manière d'écrire les instructions est toujours lécrite en Pascal.

2. Un exemple : résolution d'une équation du second degré.

Ici encore le problème est assez clair : les sorties sont
— ou bien "l'équation n'a pas de racines réelles",
— ou bien les racines de l'équation en distinguant le cas où celles-ci sont confondues.
les entrées sont les coefficients de l'équation.
Voici la liste du programme :

```
Program Second_degre;
```

```
Var
```

```
    a,b,c      : Real;  
    discriminant : Real;
```

```
Begin
```

```
    (* initialisation *)
```

```
    clrscr; (* nettoie l'écran *)
```

```
    writeln('Programme de résolution d'une équation  $ax^2+bx+c=0$ ');
```

```
    (* entrée des données *)
```

```
    writeln('donnez a,b,c');
```

```
    readln(a); readln(b); readln(c);
```

```
    (* calcul et sortie des résultats *)
```

```
    if a=0 then write('l'équation n'est pas du second degré')  
    else
```

```
        begin
```

```
            discriminant := b*b-4*a*c;
```

```
            if discriminant < 0 then write('l'équation n'a pas de  
                racines réelles')
```

```
            else if discriminant = 0 then write('l'équation a une  
                racine double', -b/(2*a))
```

```
            else write('l'équation a deux racines',  
                (-b+sqrt(discriminant))/(2*a),  
                (-b-sqrt(discriminant))/(2*a))
```

```
        end;
```

```
End.
```

On a déjà vu pourquoi l'écriture d'un programme doit tendre à être la plus lisible possible; aussi dans cette imbrication un peu compliquée d'instructions conditionnelles, nous avons essayé de rester clair en indentant le programme de façon à faire apparaître les différents groupes d'instructions.

Ce type de programme est encore bien imparfait car si vous vous trompez dans l'introduction des données, le programme avortera. Pour éviter ceci, on a intérêt à utiliser des procédures de saisies des données protégées (pour cela, nous disposons de procédures toutes faites : voir la Bibliothèque Modulog dans les chapitres suivants).

Plus encore que les instructions conditionnelles, ce sont les boucles qui nous font entrer dans le domaine de la programmation. En effet, si les ordinateurs n'étaient capables que d'exécuter une fois les opérations que nous leur décrivons dans un langage assez contraignant, ils n'auraient que bien peu d'intérêt.

A. L'instruction `while ... do ...`

La syntaxe de cette instruction est :

```
while ( expression booléenne ) do ( instruction ) ;
```

L'instruction est exécutée tant que la valeur de l'expression est vraie. Si la valeur est fausse au départ, l'instruction n'est donc jamais exécutée.

B. L'instruction `repeat ... until ...`

La syntaxe de cette instruction est :

```
repeat ( instruction ) until ( expression booléenne ) ;
```

L'instruction est exécutée jusqu'à ce que la valeur de l'expression soit vraie. L'instruction est donc toujours exécutée au moins une fois.

C. Un exemple : l'algorithme d'Euclide.

L'algorithme d'Euclide permet de trouver le plus grand diviseur commun (p.g.c.d.) de deux nombres.

L'idée de cet algorithme provient de la remarque suivante.

Si un nombre d en divise deux autres p et q alors il divise p et le reste de la division de p par q et réciproquement.

En effet, l'écriture $p = aq + r$ montre que d divise p et q si et seulement si il divise p et r .

On définit donc deux variables p et q , auxquelles sont affectées les valeurs des nombres donnés en début de programme. Ensuite, la division décrite ci-dessus est effectuée et la valeur q est affectée à la variable p , la valeur r à la variable q et on recommence. Ainsi, la suite des valeurs affectées à q est strictement décroissante et est donc nulle après un nombre fini d'opérations. A ce moment, p contient le p.g.c.d. cherché.

~~Début le programme~~

~~Program PGCD~~

~~Var p,q,r : Integer~~

~~Begin~~

~~(* Entrée des données p et q *)~~

~~writeln('calcul du p.g.c.d. de deux nombres')~~

~~writeln('donnez les deux nombres')~~

~~readln(p); readln(q);~~

~~(* Algorithme d'Euclide *)~~

~~repeat~~

~~r := p mod q;~~

~~p := q; q := r;~~

~~until q=0;~~

~~(* Sortie du résultat *)~~

~~writeln('le p.g.c.d. est ', p);~~

~~End.~~

4. Invariant de boucle.

~~Quand on réalise une boucle telle que la précédente, deux problèmes se posent :~~

~~-- la sortie de la boucle s'effectue-t-elle après un nombre fini d'opérations ?~~

~~-- à la sortie de la boucle, le résultat est-il bien celui que l'on désire ?~~

~~Pour résoudre ces deux problèmes, une bonne méthode est de construire la boucle en utilisant une propriété invariante au niveau de la sortie de la boucle telle que le test de fin de boucle implique le résultat; il faut de plus s'assurer que le test sera réalisé après un nombre fini d'itérations.~~

~~Dans le cas de l'algorithme d'Euclide, la propriété est "l'ensemble des diviseurs des valeurs affectées à p et q est inchangé". La condition q=0 implique, si elle est réalisée, que les diviseurs des nombres donnés sont ceux de p donc p est le p.g.c.d. cherché. D'autre part la suite des valeurs affectées à q est strictement décroissante donc la condition q=0 est réalisée après un nombre fini d'itérations.~~

5. Exercices.

1. Modifiez le programme de résolution des équations du second degré pour qu'il traite le cas des racines imaginaires.

2. ~~Ecrivez un programme de calcul de la factorielle d'un nombre.~~

3. ~~Ecrivez un programme de calcul des coefficients du binôme.~~

4. ~~Ecrivez un programme de décomposition d'un nombre en facteurs premiers.~~

5. ~~Ecrivez un programme permettant de tester la primalité d'un nombre.~~

6. ~~Calcul du nombre e.~~

~~On démontre que e est la somme pour n variant de 0 à l'infini des nombres $1/n!$~~

~~Calculez les sommes partielles de cette série. Déduisez-en une valeur approchée de e (Justifiez la précision obtenue).~~

7. Soit (u_n) et (v_n) les suites définies par :

$$u_0 = 4 \quad \text{et} \quad v_0 = 2\sqrt{2}$$

et les formules :

$$u_{n+1} = \frac{2u_n v_n}{u_n + v_n} \quad \text{et} \quad v_{n+1} = \sqrt{u_n v_n}$$

~~Calculez les termes successifs de ces suites. Que remarquez-vous ? Essayez de le démontrer.~~

Chapitre 3

Fonctions et procédures

Un des avantages du langage Pascal est que l'utilisateur peut définir lui-même les fonctions dont il a besoin. Par exemple, le Turbo Pascal ne possède pas de fonction calculant la puissance n -ième d'un nombre, il est cependant facile de la fabriquer. Ces fonctions s'utilisent alors comme les fonctions prédéfinies du langage.

De même, il est possible de créer de nouvelles instructions : il s'agit de ce que nous appelons des procédures.

Ces possibilités seraient bien limitées si nous ne disposions pas de bibliothèques de fonctions et de procédures qu'il est possible d'inclure dans tout programme. Ainsi le programmeur n'a pas besoin de toujours réinventer la roue comme le lui impose (presque) certains langages.

1. Les fonctions.

A. Un exemple : calcul d'une puissance n -ième.

Prenons l'exemple du calcul de la puissance n -ième d'un nombre. Ce calcul est facile à programmer, par exemple :

Si nous supposons que la variable réelle x contient déjà le nombre en question, la variable entière n l'exposant et que la variable réelle puissance est destinée à contenir le résultat, nous obtenons le morceau de programme :

```
Var      calcul: Real      ;
         indice: Integer   ;

Begin
    calcul := 1; indice := 0;

    while indice <= n do
        begin
            calcul := calcul * x;
            indice := indice + 1;
        end;

    puissance := calcul;
END;
```

Ce morceau de programme peut être transformé en une fonction de la façon qui va être décrite ci-après :

B. Déclaration.

Avant de définir une fonction, le programmeur doit lui donner un nom, nom que nous appelons identificateur comme pour les variables. Celle-ci possède des arguments de types précis appelés paramètres et un résultat (qui ne peut pas être un tableau).

Dans l'exemple précédent, les paramètres sont x et n de types réels et entiers respectivement, le résultat est puissance de type réel, la déclaration de cette fonction est alors :

```
Function puissance (x : Real; n : Integer) : Real;
```

suivi de la liste du programme précédent.

Les variables introduites au niveau de la fonction (d'identificateurs "calcul" et "indice" ici) sont appelées des variables locales car elles n'existent pas pour le programme appelant la fonction. En particulier, leurs identificateurs peuvent être réutilisés dans le corps du programme appelant sans risques de confusion. Par opposition, les autres variables sont dites globales (on restreindra au maximum l'utilisation d'objets globaux car ils rendent les programmes plus difficiles à lire).

La déclaration des fonctions se fait après la rubrique de déclaration des variables.

Les paramètres utilisés lors de la déclaration d'une fonction sont dits paramètres formels.

C. Utilisation.

Les fonctions définies par le programmeur s'utilisent comme les fonctions prédéfinies du langage. Le passage des paramètres réels (par opposition avec les paramètres formels) se fait par valeurs c'est-à-dire que les contenus des variables éventuelles utilisées pour ce faire ne sont pas modifiés. Il est possible de le faire par variables comme dans le cas des procédures mais cela est à éviter : il est de très mauvais goût qu'une fonction modifie ses arguments.

Une fonction peut en appeler une autre, elle-même en particulier. Dans ce cas, la fonction est dite récursive. On évitera d'utiliser la récursivité sans précautions car non seulement elle est délicate à programmer (il est très facile de boucler indéfiniment) mais de plus elle utilise une place énorme en mémoire.

C. Exemple de récursivité.

Par exemple, il est possible de définir la factorielle récursivement :

```
Function fact(n: Integer): Integer;  
Begin  
  if n < 1 then fact := 1  
    else fact := n * fact(n-1)  
End;
```

Le résultat est correct. Cependant un appel de fact(50) demande de stocker les appels successifs de fact(49), fact(48) ... fact(2) jusqu'à tomber sur fact(1) qui vaut 1 pour effectuer les calculs. Ce genre d'occupation inconsidérée de la mémoire est à éviter, ici il vaut mieux écrire :

```
Function fact(n: Integer): Integer;  
  
Var    calcul, indice: Integer;  
  
Begin  
  calcul := 1; indice := 1;  
  while indice <= n do  
    begin  
      calcul := calcul * indice;  
      indice := indice + 1;  
    end;  
  fact := calcul;  
End;
```

Ce qui n'utilise que deux variables entières supplémentaires.

Dans les deux cas les résultats sont très vite incorrects car les entiers sont limités à 32767, pour les améliorer on peut les définir comme réels ou mieux définir un nouveau type entier ce qui est proposé en exercice au chapitre 4.

Cependant, dans le premier cas, vous risquez de plus de saturer assez vite la mémoire, comme le montre l'exemple suivant :

```
Function calcul(n: Integer): Integer;  
Begin  
  if n < 1 then calcul := 1  
    else calcul := n + calcul(n-1)  
End;
```

Trouvez ce que fait ce programme et essayez-le pour de grands nombres ! Corrigez-le.

2. Les procédures.

page 17

De même que le programmeur peut se définir ses propres fonctions, il peut se définir ses propres instructions. Celles-ci, appelées procédures, utilisent ensuite comme les instructions prédéfinies du langage.

Un programme pascal est en fait une suite de procédures d'une page au maximum. Cette décomposition du programme en procédures reflète la décomposition de l'algorithme en sous-algorithmes.

A. Un exemple : maximum et minimum.

Prenons l'exemple du calcul des extremums de deux nombres réels. Ce calcul est facile à programmer, par exemple :

```
Begin
if a < b then
  Begin
    max := b ; min := a ;
  End
else
  Begin
    max := a ; min := b ;
  End ;
End ;
```

calcule le maximum (max) et le minimum (min) des deux nombres a et b.

Ce nouveau programme peut être transformé en une procédure.

B. Déclaration.

Avant de définir une procédure, le programmeur doit lui donner un nom, nom appelé identificateur comme dans le cas des fonctions. Celle-ci possède des arguments appelés paramètres pouvant être passés par valeurs ou par variables.

Dans l'exemple précédent, a et b sont passés par valeurs (on ne désire pas que la procédure les modifie), min et max par variables car la procédure doit les modifier. Ce qui donne :

```
Procedure extremum (a:Real; b:Real; Var max:Real; Var min:Real) ;
```

suit de la liste du programme précédent.

Les procédures sont déclarées en même temps que les fonctions.

C. Utilisation.

~~Les procédures définies par le programmeur utilisent comme les instructions prédéfinies du langage.~~

~~Une procédure peut en appeler une autre si elle est déclarée avant elle-même.~~

3. Utilisation d'une bibliothèque.

~~Pour utiliser une bibliothèques de procédures et de fonctions, il faut que celle-ci soit suffisamment documentée pour que l'on puisse leur passer leurs paramètres correctement et prendre en compte l'environnement qui leur est utile c'est-à-dire les procédures ou les fonctions, ainsi que les variables globales qu'elles utilisent.~~

~~Les bonnes bibliothèques donnent toujours ces renseignements ainsi que des indications sur les algorithmes utilisés.~~

A. Un exemple : utilisation d'un évaluateur de fonctions.

~~En guise d'exemple, nous montrons l'utilisation d'une procédure permettant d'entrer une fonction au clavier directement sous forme algébrique.~~

~~Cette procédure provient de la bibliothèque Modulog, bibliothèque de procédures de l'A.L.E. Sup disponible au lycée. Elle a beaucoup de possibilités que nous ne détaillerons pas ici, voici seulement comment l'utiliser pour calculer des valeurs d'une fonction entrée directement au clavier.~~

~~Grace à cette bibliothèque, nous disposons (en particulier) d'une procédure et de deux fonctions :~~

~~entre_fct sans argument mais permettant d'entrer la fonction $f(x)$,~~

~~$f(x)$ dont l'argument et la valeur sont réels,~~

~~entre_nb sans argument dont la valeur est réelle.~~

~~Nous disposons également d'une fonction fin_boucle sans argument à valeur booléenne permettant d'arrêter une boucle d'entrée de données : sa valeur est True si l'arrêt est demandé par l'utilisateur.~~

B. Calcul de valeurs d'une fonction.

page 19

Voici un programme utilisant ces procédures :

Program Calcul_ft;

(*#fonction.lib*)(*#ioreal.lib*)(*#ilogf(X),int*)

(* Ces instructions permettent d'inclure dans le programme les fichiers donnant accès à la procédure entre_ft et aux fonctions f, valeur et fin_boucle *)

Var x : Real;

Begin

clrscr;

entre_ft;

repeat

Begin

writeln('Entrez une valeur');

x := entre_nb;

writeln(f(x));

End;

until fin_boucle;

writeln('Au revoir');

End.

4. Un exemple : Résolution d'une équation par dichotomie.

Nous considérons une fonction f continue sur un segment $[a, b]$ à valeurs dans $\mathbb{R}$ changeant de signe une fois et une seule sur $[a, b]$. Nous savons alors que l'équation $f(x) = 0$ a une solution unique sur $[a, b]$.

A. Exposé de la méthode.

Pour déterminer la solution avec une précision epsilon donnée, la méthode consiste à couper l'intervalle $[a, b]$ en deux et choisir le sous-intervalle contenant la solution puis de recommencer jusqu'à ce que la longueur de l'intervalle obtenu soit inférieure à epsilon. La solution est alors l'une des extrémités de l'intervalle final.

Remarque : si l'équation a plusieurs zéros avec changement de signe, la méthode donne l'un des zéros. Si l'équation ne change pas de signe alors le résultat sera l'une des extrémités de l'intervalle, bien entendu il sera alors erroné.

B. Structure du programme.

Nous définissons ainsi une fonction d'arguments a, b et ϵ de type réel et de résultat la solution de type réel d'où la déclaration :

```
function dichotomie(a:Real; b:Real; eps:Real):Real;
```

Cette fonction utilise une autre fonction déterminant si deux nombres sont de même signe :

```
function signes_egaux(x:Real; y:Real):Boolean;
```

dont le résultat est vrai si les signes sont égaux, faux sinon.

```
Program Equation;
```

```
(*) (fonction.lib) (coreal.lib) (logf(x).int)
```

```
Var min, max, eps :Real;
```

```
Function signes_egaux(x:Real; y:Real):Boolean;
```

```
Begin
```

```
signes_egaux := ((x < 0) and (y < 0))  
or ((x >= 0) and (y >= 0));
```

```
End;
```

```
Function dichotomie(min:Real; max:Real; eps:Real):Real;
```

```
Var milieu :Real;
```

```
f_min :Real;
```

```
Begin
```

```
f_min := f(min);
```

```
while max - min > eps do
```

```
Begin
```

```
milieu := (max + min) / 2;
```

```
if signes_egaux(f_min, f(milieu))
```

```
then min := milieu
```

```
else max := milieu;
```

```
End;
```

```
dichotomie := min;
```

```
End;
```

```

Begin
  clrscr; (*nettoie l'écran*)
  entre_fct;
  repeat
    Begin
      writeln('donnez les bornes et la précision');
      min:=entre_nb; max:=entre_nb; eps:=entre_nb;
      writeln('la solution est', dichotomie(min,max,eps));
    End;
  until fin_boucle;
  writeln('Au revoir');
End.

```

5. Exercices.

1. Résolution d'une équation.

Programmez la méthode de balayage consistant à essayer toutes les valeurs de l'intervalle donné avec un pas epsilon donné. Comparez avec la méthode de dichotomie.

Calcul d'une intégrale.

2. Méthodes à un pas.

Créez les fonctions calculant une valeur approchée de l'intégrale de f entre les bornes a et b avec les méthodes à un pas suivantes :

* méthode des rectangles $(b-a) \cdot f((a+b)/2)$

* méthode de Simpson $((b-a)/6) [f(a) + 4 f((a+b)/2) + f(b)]$

3. Méthodes à pas constant.

Utilisez les fonctions écrites lors de la résolution de l'exercice 1 pour calculer l'intégrale en appliquant une méthode à un pas à chaque intervalle d'une subdivision en n parties égales.

4. Méthodes à pas variables.

On calcule l'intégrale avec une méthode à un pas sur l'intervalle entier puis sur les deux demi-intervalles, si la différence entre les deux résultats est inférieure à epsilon donné, on dit qu'il s'agit du résultat sinon on recommence avec chacun des deux demi-intervalles et epsilon divisé par deux.

Utilisez cette idée pour programmer le calcul d'une intégrale (bien entendu, vous définirez une fonction récursive).

5. ~~Calcul d'une dérivée.~~

~~Programmez le calcul approché d'une dérivée.~~

~~Vous comparerez l'utilisation d'une dérivée latérale et celle d'une dérivée centrale~~

$$\frac{f(x+h) - f(x)}{h} \quad \text{et} \quad \frac{f(x+h) - f(x-h)}{2h}$$

6. ~~Estimation de l'ordre d'un infiniment petit.~~

~~Programmez le calcul de l'estimateur de Daniel Hervé~~

$$\frac{\ln \frac{f(kx)}{f(x)}}{\ln k}$$

~~pour x proche de 0 (0.1 par exemple) et k proche de 1 (1.1 par exemple).~~

~~Utilisez-le pour évaluer l'ordre de~~

$$\text{tg}(\sin x) - \sin(\text{tg} x)$$

~~au voisinage de 0.~~

~~Expliquez.~~

~~Extremum d'une fonction.~~

7. ~~Programmez la recherche de l'extremum d'une fonction par balayage de l'intervalle $[a, b]$ donné (le balayage consiste à essayer toutes les valeurs de l'intervalle avec un pas h donné).~~

8. ~~Méthode par dichotomie.~~

~~La méthode de dichotomie pour la recherche d'un extremum revient à couper l'intervalle donné en quatre, et à comparer les valeurs de la fonction aux extrémités de ces intervalles. On en déduit que l'extremum appartient à l'un des trois intervalles de longueurs moitié ainsi déterminés et on détermine cet intervalle.~~

~~Comparez avec la méthode précédente.~~

Les tableaux

Pour l'instant, nous ne pouvons pas définir de types correspondants aux objets mathématiques usuels comme les polynômes ou les matrices. Pour cela, nous disposons des types tableaux.

1. Les tableaux.

Un tableau est une variable constituée d'autres variables (ses éléments) ayant un type bien défini (par exemple : Integer, Real ou Boolean) auxquelles on peut accéder au moyen d'un ou de plusieurs indices (nous nous limiterons aux indices entiers).

A. Un exemple très simple.

On écrit par exemple :

```
Type      Vecteur = Array [1..3] of Real;
```

Ce qui déclare le type vecteur comme un tableau de trois réels.

Nous pouvons alors déclarer une variable *V* comme étant du type vecteur :

```
Var      V : Vecteur;
```

Pour accéder à l'élément d'indice *i*, il suffit d'écrire *V[i]*. De façon plus générale, on pourra accéder à *V[indice]* si indice est une variable entière dont le contenu est compris entre 1 et 3.

Par exemple, voici l'écriture d'une fonction permettant de calculer le produit scalaire de deux vecteurs de l'espace usuel :

```

Function Prod_Scal (U : Vecteur ; V : Vecteur) : Real ;
Var indice : Integer ;
    somme : Real ;

Begin
    indice := 0 ; somme := 0 ;
    while indice < 3 do
        Begin
            indice := indice + 1 ;
            somme := somme + U[indice] * V[indice] ;
        End ;
    prod_scal := somme ;
End ;

```

Nous obtenons une programmation légèrement plus simple en utilisant une boucle à compteur c'est-à-dire l'instruction `for ... to ... do ...` dont la syntaxe est semblable à celle de `while ... do` :

```

Function Prod_Scal (U : Vecteur ; V : Vecteur) : Real ;
Var indice : Integer ;
    somme : Real ;

Begin
    somme := 0 ;
    for indice := 1 to 3 do
        somme := somme + U[indice] * V[indice] ;
    prod_scal := somme ;
End ;

```

B. Tableaux à une dimension.

En partant de cet exemple, nous pouvons également définir un type vecteur à 100 éléments, il suffit d'écrire :

```
Type Vecteur == Array[1..100] of Real ;
```

Mais, il est plus prudent de déclarer d'abord une constante longueur_vecteur :

```
Const longueur_vecteur == 100 ;
```

pour ensuite déclarer :

```
Type Vecteur == Array[1..longueur_vecteur] of Real ;
```

Malheureusement, et c'est bien regrettable, il n'est pas possible de déclarer longueur_vecteur comme une variable que l'on pourrait ensuite affecter après une lecture au clavier par exemple. Mais, bien entendu, on peut tourner cette difficulté : voir le paragraphe 2.

Nous pouvons alors modifier la fonction précédente :

```
Function Prod_Scal (U : Vecteur ; V : Vecteur) : Real ;
  Var indice : Integer ;
      somme   : Real   ;

Begin
  somme := 0 ;
  for indice := 1 to longueur_vecteur do
    somme := somme + U[indice] * V[indice] ;
  prod_scal := somme ;
End ;
```

Où vous pouvez voir que les modifications ne sont pas trop difficiles à opérer du fait du style de programmation employé.

C. Tableaux à deux dimensions.

Nous pouvons également définir des tableaux à deux indices, par exemple pour définir un type matrice carrée d'ordre 3, on peut écrire :

```
Const ordre_mat := 3 ;

Type Matrice = Array[1..ordre_mat, 1..ordre_mat] of Real ;

La multiplication est réalisée par une procédure très facile à écrire :
```

```
Procédure Prod_Mat (A : Matrice ; B : Matrice ; Var C : Matrice) ;
  Var ligne, colonne, indice : Integer ;
      somme                   : Real   ;

Begin
  for ligne := 1 to ordre_mat do
    Begin
      for colonne := 1 to ordre_mat do
        Begin
          somme := 0 ;
          for indice := 1 to ordre_mat do
            somme := somme + A[ligne, indice] * B[indice, colonne] ;
          C[ligne, colonne] := somme ;
        End ;
      End ;
    End ;
  End ;
```

Remarque :

Il est également possible de définir le type Matrice comme un tableau à une dimension de vecteurs, ce qui donne :

```
Const dimension == 100;
```

```
Type Vecteur == Array[1..dimension] of Real;
```

```
Type Matrice == Array[1..dimension] of Vecteur;
```

Nous accédons alors directement aux vecteurs colonnes (ou lignes, si nous préférons les considérer ainsi) de la matrice. La programmation du produit matriciel ou de la méthode de Gauss en sera changée voir l'exercice 10.

2. Problème de la dimension.

Pour disposer de matrices d'ordres non définis à l'avance, le plus simple est de définir un ordre maximum en constante: `Ordre_Mat_Max` par exemple puis de travailler systématiquement sur ces matrices. Par exemple voici une adaptation du programme précédent dans cet esprit:

```
Procedure Prod_Mat(A: Matrice; B: Matrice;
```

```
Var C: Matrice; ordre_mat: Integer);
```

```
Var ligne, colonne, indice: Integer;
```

```
    somme: Real;
```

```
Begin
```

```
    for ligne == 1 to ordre_mat do
```

```
        Begin
```

```
            for colonne == 1 to ordre_mat do
```

```
                Begin
```

```
                    somme == 0;
```

```
                    for indice == 1 to ordre_mat do somme == somme +
```

```
                        A[ligne, indice] * B[indice, colonne];
```

```
                    C[ligne, colonne] == somme;
```

```
                End;
```

```
            End;
```

```
        End;
```

Le produit obtenu sera bien le résultat escompté: le seul défaut de la méthode est la perte inconsiderée de mémoire.

Bien entendu, on aura déclaré au préalable:

```
Const Ordre_Mat_max == 100;
```

```
Type Matrice == Array[1..Ordre_Mat_max, 1..Ordre_Mat_Max] of Real;
```

```
Var Ordre_Mat: Integer;
```