

## CHAPITRE 7

### FICHIERS, POINTEURS ET LISTES

#### Fichiers à accès direct en turbo-pascal

En turbo-pascal sur PC le type "file of ..." permet les fichiers de longueur variable, il faut pour utiliser des fichiers sur disquette connaître les ordres de création "rewrite", d'ouverture "reset" de fermeture "close" etc...

On se limite volontairement dans le programme de carnet d'adresses, aux opérations élémentaires sur un fichier à accès direct (chaque élément se trouve à une adresse précise), aller chercher une fiche, en créer une nouvelle, et lire la totalité. Les différentes instructions de manipulation de fichiers sont commentées au fur et à mesure dans le programme.

```

program adresses ; { Fichier à accès direct }
    type individu = record nom, prenom : string [20];
                        adresse : string [80]; telephone: string [12] end;
    fichier = file of individu; { On définit deux types d'objets, les individus et les fichiers }
    var G : fichier; X : integer ;

procedure creation (var F : fichier); { ne servira qu'une seule fois }
    begin rewrite (F); close (F) end;

procedure ecriture (var F : fichier); { Ecrit une ou plusieurs nouvelles fiches dans F }
    var A : individu ; C : char;
    begin reset (F); C := 'O';
    seek (F, filesize (F)); { permet de se positionner sur la fin du fichier en cherchant (seek) dans F
la dernière position (filesize(F) donne le nombre de fiches)."filesize" est inconnu sur Macintosh }
    with A do while C = 'O' do begin
        write ('NOM: '); readln (nom); write ('PRENOM: '); readln (prenom);
        write ('ADRESSE: '); readln (adresse); write ('TEL: '); readln (telephone);
        write (F, A); { Ecriture dans le fichier F et non sur l'écran }
        write ('AUTRE ? (O/N) '); readln (C) end;
    { With A do... est une instruction dont il n'a pas été question jusqu'à présent, elle permet de ne pas répéter à
chaque fois A.nom, A.prenom etc... }
    close (F) end; { fin de la procédure d'écriture }

procedure edition (A : individu); {réalise une édition à l'écran de l'individu A }
    var I : integer;
    begin writeln (A.nom,' ', A.prenom,' ', A.adresse, ' TEL:', A.telephone);
    for I := 1 to 80 do write ('-') {Place une ligne séparatrice de tirets} end;

procedure lecture (var F : fichier); {Lecture complète du fichier F}
    var A : individu ;
    begin reset (F);
    repeat read (F,A); {lecture depuis F sur la disquette et non depuis le clavier, "read" et "write"
avancent toujours d'un cran la position courante dans le fichier, après leur exécution, cela est très commode pour
la procédure présente }
        write ( filepos (F),' '); edition (A)
    until eof ; { répéter jusqu'à la fin du fichier (end of file) }
    writeln ('Il y a ', filesize (F), ' fiches. '); close (F) end;

```

```

begin assign (G, 'ADRESSES.DAT'); { Permet d'assigner à la variable globale G le nom précis du fichier on
peut prévoir de demander le nom (commande propre au turbo-pascal sur PC)}
      writeln (' Fichier d adresses. ');
      write (' 0: Création  1: Lecture complète  2: Nouvelle fiche '); readln (X);
      case X of
        0      : creation (G);
        1      : lecture (G);
        2      : ecriture (G) end; { "recherche" et "destruc" restent à écrire }
end.

```

Ce programme ne fournit donc qu'une initiation, naturellement les gros fichiers posent bien d'autres problèmes mais on trouvera en exercice des idées d'améliorations.

### Variables dynamiques ou pointeurs

Jusqu'à présent lorsqu'on définissait une variable X entière en l'affectant de la valeur 3, on avait de façon "statique" durant toute l'exécution du programme, une place déterminée en mémoire où étaient rangés le nom X et la valeur 3 (éventuellement modifiée). Pour une variable dynamique, sa création ou destruction pourra être faite durant l'exécution suivant les besoins, on ne pourra y accéder que par une "variable pointeur" contenant son adresse en mémoire.

Un pointeur est défini par exemple par `var Y : ^integer;` pour un pointeur sur un entier, et le contenu de ce vers quoi elle pointe sera accessible par `Y^` qui est donc un entier.

La procédure de création d'un pointeur est `new (Y)`, cette nouvelle variable Y est alors rangée dans une pile spéciale appelée "tas".

Sa destruction est opérée par `dispose (Y)`, en ce cas sa place dans le tas est récupérée.

Le fait de ne pointer sur rien est décidé (quelquesoit le type du pointeur) par `Y := nil`

Exemple, si `var X, Y : ^integer;` a été déclaré,

Les trois séquences d'instructions `new (X); X^ := 5; Y := X; write (Y^)`

`new (X); new (Y); X^ := 5; Y := X; write (Y^)`

`new (X); X^ := 5; Y := X; dispose (X); write (Y^)` donneront toutes 5, par contre :

`new (X); X^ := 5; Y := X; Y^ := 3; write (X^);` donnera 3.

### Utilisation pour la construction d'une liste chaînée

On devra pour un exemple minimal parler de "doublets" c'est à dire d'une partie information et d'une partie "pointeur" pour aller chercher le doublet suivant. Ainsi pour une liste d'entiers on définira deux types (se définissant récursivement) :

```

type   ptr = ^doublet ;
       doublet = record numero : integer; suivant : ptr ;

```

### Exemple : tenue à jour d'une course

On souhaite en temps réel, à chaque arrivée de coureur, entrer ce coureur (son numéro de dossard, son nom et le temps qu'il vient de faire), l'insérer dans la liste de ceux qui sont déjà arrivés et réafficher le classement provisoire.

On dispose donc en permanence de la liste des coureurs ayant concouru, triée par ordre de temps croissant.

```

type   ptr = ^doublet; { doit être défini avant }
       coureur = record num : integer; nom : string[12]; temps : real end;
       doublet = record individu : coureur; suivant : ptr end;
var tete, nouveau : ptr;

```

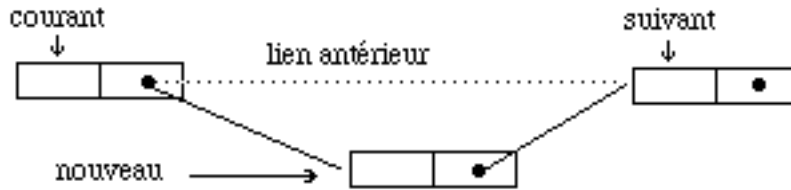
**procédure arrivée** (var nouveau : ptr); { crée un nouvel individu à son arrivée }

```

begin   nouveau^.suivant := nil;
        write (' Quel numéro '); readln (nouveau^.individu.num);
        write (' Quel temps '); readln (nouveau^.individu.temps);
        write (' Quel nom '); readln (nouveau^.individu.nom)

```

**end;**



```

procedure insertion (var tete : ptr; nouveau : ptr); { suivre la figure ci-dessus }
  var courant : ptr; { pointeur local servant à parcourir la liste }
begin
  courant := tete;
  if tete = nil then tete := nouveau { cas du tout premier concourant }
  else
    begin
      while (courant^.suivant <> nil) and
            (courant^.suivant^.individu.temps < nouveau^.individu.temps)
      do courant := courant^.suivant;
      if courant = tete then begin nouveau^suivant := tete ; tete := nouveau end
      else begin nouveau^suivant := courant^.suivant ;
            courant^.suivant := nouveau end
    end { nouveau a été intercalé entre courant et le suivant du courant }
end;

```

Autre version :

```

procedure insertion (var liste : ptr; nouv : ptr);
  var pred : ptr; { on s'intéresse au précédent "pred" }
begin new (pred); pred^.suivant := liste; liste := pred;
  while (pred^.suivant <> nil) and (nouv^.temps > pred^.suivant^.temps)
  do pred := pred^.suivant ;
  nouv^.suivant := pred^.suivant;
  pred^.suivant := nouv;
  liste := liste^.suivant { on rétablit les choses }
end;

```

```

procedure resultat (tete : ptr);
  var courant : ptr;
begin
  courant := tete; writeln ('Voici la liste des coureurs arrivés ');
  while courant <> nil do
    begin writeln (courant^.individu.num, ' ', courant^.individu.nom, ' ',
                  courant^.individu.temps); courant := courant^.suivant end
end;

```

Le programme sera (avec les déclarations d'usage) :

```

new (tete); while fini = 'n' do begin arrivee (nouveau); insertion (tete, nouveau); resultat (tete);
write ('fini ? (o/n) '); readln (fini) end

```

### Exemple de construction d'un arbre binaire

Pour définir une arborescence de caractères où chaque noeud possède un fils droit et un fils gauche, il suffit de déclarer :

```

type
  ptr = ^triple;
  triplet = record noeud : char; filsdroit, filsgauche : ptr end;

```

Sa construction pourra se faire (dans l'ordre racine-gauche-droite) par :

```

procedure construc (var A : ptr);
  var chg, chd : char;
begin
  write ('Pour ', A^.noeud, ' Fils gauche '); read (chg); new (A^.filsgauche);
  if chg = chr(13) then A^.filsgauche := nil
  else A^.filsgauche^.noeud := chg ;
  write (' Fils droit '); readln (chd); new (A^.filsdroit) ;
  if chd = chr(13) then A^.filsdroit := nil else A^.filsdroit^.noeud := chd;
  if chg <> chr(13) then construc (A^.filsgauche);
  if chd <> chr(13) then construc (A^.filsdroit) end;

```

```

begin new(A); write ('Donnez la racine '); readln (A^.noeud); construc (A) end.

```

Le parcours racine-gauche-droite d'une arborescence se fera par :

**procedure parcours** (A : ptr);

begin if A <> nil then begin write (A^.noeud); parcours (A^.filsgauche); parcours (A^.filsdroit) end end;

### Problème général du parcours avec retour en arrière

Dans une arborescence de racine n (en fait à chaque noeud n se pose le même problème), les paramètres intervenant sont le chemin ch suivi au dessus de n (donc la liste vide si n est vraiment la racine), l'ensemble des solutions sol déjà obtenues, l'ensemble "fr" des frères de droite de n et la liste "efr" des ensembles de frères non encore explorés correspondant à chacune des étapes du chemin ch. (Ici n n'est pas dans ch, par n&ch, nous entendons le chemin débutant par n et se poursuivant par ch.)

La procédure générale, en parcourant l'arborescence dans le sens racine-gauche-droite, est:

**Parcours (n, fr, ch, efr, sol)**

```
si feuille (n) alors si solution (n1&ch)
                    alors parcours (premier frère de n, queue de fr, ch, efr, sol ∪ {ch})
                    si fr vide alors si ch vide alors fin, on renvoie sol
                                sinon parcours (premier de ch,
                                                premier ensemble de efr,
                                                queue (ch), queue (efr), sol)
                                sinon parcours (premier frère, queue de fr, ch, efr, sol)
                    sinon parcours (premier fils de n, autres fils, premier fils de n & ch, fr & efr)
```

Ecriture sous forme d'une fonction booléenne au cas où on ne cherche que la première solution:

```
f(n) = "vrai ssi il y a une solution passant en n"
      = [(n est une feuille) et (n est l'aboutissement d'une solution)]
        ou [il existe un fils i de n tel que f(i)]
      = si feuille (n) alors si solution(n) alors fin (succès) f(n) ← vrai
        sinon fin (échec) f(n) ← faux
        sinon drap ← faux
              k ← 1
              répéter drap ← f(fil de n)
                    incrémenter (k)
              jusqu'à drap ou k > card(fil(n))
              f(n) ← drap
```

On stoppe ainsi à la première solution qui dépend bien sûr de l'ordre de rangement des fils.

### Exemple de fichiers séquentiels en pascal standard : fusion de deux fichiers croissants existants en un seul.

Les instructions sont :

```
rewrite (F) pour la création
reset (F) pour l'ouverture
get (F) avancée en lecture du pointeur
put (F) avancée en écriture
read (F, X) affectation de l'élément courant de F sous le nom de X et avancée
write (F, X) écriture dans F de X à la position courante et avancée du pointeur
```

**program fusion** (F1, F2, F3);

var F1, F2, F3 : file of integer;

begin reset (F1); reset (F2); rewrite (F3);

while not (eof(F1)) and not (eof(F2)) do begin if F1^ < F2^ then read (F1, F3^) else read (F2^, F3^);  
put (F3) end;

while not (eof(F1)) do begin read (F1, F3^); put (F3);

while not (eof(F2)) do begin read (F2, F3^); put (F3) end.

Mais chacune des suites d'instructions lecture-écriture peut être modifiée en write (F3, F1^); get (F1) par exemple.

**7-1° Procédure de lecture complète d'un fichier F en accès direct**

```

procedure lecture (var F : fichier); { Lecture complète du fichier F }
  var A : individu ;
  begin reset (F); seek (F, 1);{ on ne considère pas la fiche n° 0 }
  repeat write (filepos (F),' '); read (F, A); edition (A)
  until eof (F);
  writeln ('Il y a ', filesize (F)-1,' fiches. ');
  close (F) end;

```

**7-2° Ecrire une procédure intercalant X dans le fichier F en accès direct à la place I.**

```

procedure intercaler (X: individu; var F : fichier; I : integer);
  { intercale l'individu X à la place I dans le fichier F ouvert prévoir le cas I=n+1 }
  var J : integer; A : individu;
  begin for J := filesize (F) downto I do
        begin seek (F, J-1); read (F, A); write (F,A) end;
  { repousse tous les éléments de F d'un cran à partir de I }
  seek (F, I) ; write (F, X) end;

```

**7-3° Ecrire une procédure de destruction d'une fiche, en faisant glisser toutes les fiches de numéros supérieurs d'un cran (programme dit de ramasse-miettes), utiliser la procédure Pascal "truncate" (F) afin de supprimer réellement la ou les dernières fiches.**

```

procedure destruc (var F : fichier);
  var I, K : integer; A : individu;
  begin lecture (F);
  write ('Quel numéro '); readln (I) ; reset (F) ; seek (F, I);
  for K := I+1 to filesize (F) - 1 do { ramasse miettes }
        begin seek (F, K); read (F, A); seek (F, K-1); write (F, A) end;
  truncate (F); { le dernier élément est alors supprimé }
  lecture (F) end;

```

**7-4° Dans le cas où l'insertion se fait suivant l'ordre alphabétique, écrire une fonction à deux paramètres, un mot N et un fichier F, renvoyant le rang où devrait s'insérer un individu de nom N.**

On peut utiliser pour cela une fonction dite de "hachage" calculant approximativement cette position en fonction de la première lettre, puis par comparaison on avance ou on recule dans le fichier.

```

function rechercheplace : (N : individu ; var F : fichier ) : integer ;
  { Donne le rang alphabétique où doit s'insérer A , F est ouvert }
  var I, K : integer ; A : individu ;
  begin K := filesize (F);
  I := trunc ( K*(ord (N.nom [1]) - 64) / 26 ); { ici on peut trouver une fonction plus adaptée aux noms français }
  if I = 0 then I := 1;
  if K = 0
    then rechercheplace := 1
    else begin seek (F, I); read (F, A);
        if A.nom < N.nom
          then if I = K
                then rechercheplace := K + 1
                else
                  begin
                    repeat read (F, A); I := I+1;
                    until eof or (N.nom < A.nom);
                    rechercheplace := I end
          else if I = 1
                then rechercheplace := 1
                else begin
                    repeat I := I - 1; seek (F, I) ; read (F, A)
                    until (I = 0) or (A.nom < N.nom) ;
                    rechercheplace := I + 1 end
        end;
  end end;

```

**7-5°** Prévoir une procédure d'insertion qui, après chaque nouvelle fiche obtenue, la place automatiquement dans le fichier à sa place suivant l'ordre alphabétique.

Solution faisant appel à "rechercheplace" :

```

procedure demande (var F : fichier);
var A : individu; C: char;
begin C := 'O'; reset (F);
with A do while C = 'O' do begin
    write ('NOM:'); readln (nom);
    write ('PRENOM:'); readln (prenom);
    write ('ADRESSE:'); readln (adresse);
    write ('TEL:'); readln (telephone);
    intercaler (A, F, rechercheplace (A, F) );
    write ('Autre fiche ? (O / N)'); readln (C)
end;
close (F) end;

```

**7-6°** Ecrire une procédure de recherche, prévoyant un tableau pour les homonymes contenant les prénoms et les numéros de fiches. Ecrire une procédure de modification des différentes rubriques pour une fiche. "modif" peut très bien être définie comme une sous-procédure à l'intérieur de "recherche".

```

procedure recherche (var F : fichier);
    type index = record    prenom : string [20];
                          num : integer end; { n° de fiche }
    var X, Y: string [20] ; I, J : integer ; A : individu;
        T : array [1..10] of index; { T contient les homonymes }
    procedure modif (var F : fichier; I : integer);
        var K : integer;
    begin write ('Modif du nom 1, prénom 2, adresse 3, tel 4 '); readln (K);
    case K of 1:  { à compléter }   end {du case}
end;

begin {de recherche} write ('Quel nom :'); readln (X); J := 0;
reset (F); for I := 1 to filezise (F) do
    begin read (F, A);
    if A.nom = X then    begin edition (A); J := J+1;
                        T[J].prenom := A.prenom ;
                        T[J].num := I end end;
case J of
    0 : writeln ('Ne figure pas au fichier') ;
    1 : modif (F, T[1].num);
else    begin write ('Quel prenom :'); read (Y);
        for I := 1 to J do if Y=T[I].prenom    then modif (F, I)
                           else recherche (F)
        end;
end; {du case}
close (F) end;

```

**7-7° Recherche par dichotomie** d'un élément X dans un tableau [1..m] d'individus triés suivant leur âge.

On suppose qu'"individu" est un type défini par :

```

type    individu = record nom, prenom : string[15]; age, telephone : integer end;
        table = array[1..m] of individu;

```

On définit alors une fonction rang donnant la place de X dans T s'il y est, et la place qu'il devrait avoir sinon.

```

function rang (X : individu; T : table; i, j : integer) : integer; {rang de X entre les indices i et j compris}
begin
    if X.age < T[i].age then rang := i - 1
    else if X.age = T[i].age then rang := i
    else if X.age = T[j].age then rang := j
    else if X.age > T[j].age then rang := j + 1
    else if j = i + 1 then rang := j
    else if T[(i + j) div 2].age < X.age then rang := rang(X, T, (i + j) div 2 + 1, j - 1)
    else rang := rang(X, T, i + 1, (i + j) div 2)
end;

```

On appellera rang (X, T, 1, m).

**7-8°** Procédure sauvant un fichier séquentiel avec des villes et coordonnées à partir d'un tableau.

```

const nv = 162; {nb de villes};
type
    entier = 0..280;
    ville = record nom : string [20]; x, y : entier end;
    fvilles = file of ville;
var v : array [1..nv] of ville;    {on suppose que le tableau v est bien rempli}

procedure sauver (n : entier; nom : string); {sauve le tableau v de 1 à n dans "nom"}
    var j : entier; fichier : fvilles;
    begin rewrite (fichier, nom); for j := 1 to n do write (fichier, v[j]) ; close( fichier) end;

```

**7-9°** Procédure chargeant un tableau depuis un fichier séquentiel de villes (noms et coordonnées).

```

procedure charger (nom : string ) {charge le fichier de villes "nom" dans v};
    var i : entier; fichier : fvilles;
    begin reset (fichier, nom); i := 0;
        while not (eof(fichier)) do begin i := i+1; read (fichier, v[i]) end
    end;
begin charger ( 'pays'); for i := 1 to nv do write (v[i].nom) end.

```



**7-10°** Ecrire un Lisp en pascal (on se limite à des listes plates d'entiers).

Il est nécessaire de connaître les bases du Lisp (chapitres suivants) pour comprendre les opérations relatives à la structure de liste qui suivent.

```

type      liste = ^doublet;
          doublet = record tete : integer; queue : liste end;

function car (l : liste) : integer; { donne le premier élément de L }
  begin if l = nil then writeln ('erreur dans car') else car := l^.tete end;

function cdr (l : liste) : liste; { donne la liste L privée de son premier élément }
  begin if l = nil then cdr := nil else cdr := l^.queue end;

function cons (n : integer; l : liste) : liste;
  var p : ptr; begin new(p); p^.tete := n; p^.queue := l; cons := p end;

function vide (L : liste) : liste;
  begin vide := (L = nil) end;

function longueur (l : liste) : integer;
  begin if l = nil then longueur := 0 else longueur := 1 + longueur(cdr(l)) end;

function concat (l1, l2 : liste) : liste; { concaténation de deux listes }
  begin if l1 = nil then concat := l2 else concat := cons (car(l1), concat (cdr (l1), l2)) end;

```

**7-11°** Tri par fusion pour les listes précédentes. Une liste d'entiers sera triée par fusion si ses sous-listes d'éléments de rangs pairs et impairs sont triées séparément suivant la même méthode, puis fusionnées.

Nous adoptons une stratégie consistant à séparer une liste en deux listes constituées par les éléments d'ordre impairs ou pairs, de façon à ne parcourir la liste qu'une seule fois. Puis, le tri proprement dit se fait sur ces deux parties qui sont ensuite fusionnées.

```

function fusion (L1, L2 : liste) : liste;
  begin  if vide (L1) then fusion := L2
         else if vide (L2) then fusion := L1
         else if L1^.tete < L2^.tete then fusion := cons (L1^.tete, fusion (L1^.queue, L2))
         else fusion := cons (L2^.tete, fusion (L1, L2^.queue)) end;

procedure separ (pair : boolean; var LP, LI, LR : liste) ; { construit deux listes LP et LI à partir de L }
  begin if not (vide (LR)) then
        if pair  then separ (false, cons (LR^.tete, LP), LI, LR^.queue)
        else separ (true, LP, cons (LR^.tete, LI), LR^.queue)
  end;

function trifus (L : liste) : liste ; { produit la liste triée à partir de L }
  var LP, LI : liste;
  begin if vide (L)  then trifus := L
        else  begin new (LP); new (LI) ; LP := nil; LI := nil;
              separ (true, LP, LI, L); trifus := fusion (trifus (LP), trifus (LI)) end end;

```

**7-12°** Lire un fichier séquentiel F en construisant au fur et à mesure deux fichiers FP et FI contenant dans le même ordre les éléments de F respectivement d'ordres pairs et impairs.

**7-13° Petit système expert** : on veut toutes les propositions déduites d'une liste d'axiomes et d'une liste d'implications. Par exemple si A, C, D et les règles  $A \& C \rightarrow B$ ,  $B \& F \rightarrow E$ ,  $B \& F \rightarrow I$ ,  $B \& D \rightarrow F$ ,  $A \& G \rightarrow K$  on aura A, B, C, D, E, F, I.

```

program minisystemexpert;
const m = 10; {10 règles au maximum}
type
    ptr = ^doublet;
    doublet = record prop : char; suivant : ptr end;
    {ainsi est définie une liste de propositions}
    implic = record conclusion : char; premisses : doublet end;
    {une règle est une suite de propositions (les hypothèses), et la conclusion}
var nr : integer; tf : array [A .. Z] of boolean; tr : array [1 .. m] of implic;

procedure entreprop (var np : integer; var tp : array[A..Z] of boolean);
{entrée de np propositions (mises à vrai), les autres lettres étant mises à faux, tp contiendra donc les valeurs de
vérité de tous les faits à chaque instant}
procedure entreeimplic (var ni : integer; var ti : array[1..m] of implic);
{on pourrait aussi chaîner les implications et même les mélanger aux axiomes}
    var tete : ptr; j : integer; lettre : [A..Z];
begin write ('combien de règles ? '); readln(ni);
for j := 1 to ni do begin write('implication n° ', j, ' entrez les prémisses (des lettres) séparées par "return" ');
    new (tete); tete := nil; readln (lettre);
    repeat tete^.suivant := tete;
        tete^.premisses := lettre;
        readln (lettre)
    until lettre = "";
    write (' conclusion '); readln (lettre);
    ti[j].conclusion := lettre; ti[j].premisses := tete
end;

end;

procedure vue (regle : implic);
    var suiv : ptr;
begin write ('On applique la règle '); new (suiv); suiv := regle.premisses;
while suiv <> nil do begin write (suiv^.prop, ' '); suiv := suiv.suivant end;
write ('--> ', regle.conclusion)
end;

function declench (regle : implic; t : array [A..Z] of boolean) : boolean;
    var suiv : ptr; poss : boolean ;
begin poss := true; suiv := regle.premisses ;
repeat
    poss := t[suiv^.prop];
    suiv := suiv^.suivant
until not (poss) or (suiv = nil);
declench := poss end;

procedure chainage (var t : array[A..Z] of boolean; ni : integer; ti : array[1..m] of implic);
    var drap : boolean ; j : integer ;
begin
repeat
    drap := false;
    for j := 1 to ni do
        if not (t[ti[j].conclusion]) and declench (ti[j], t) then
            begin t[ti[j].conclusion] := true;
                drap := true; vue (ti[j]) end
    until not (drap) end;

procedure sortie (t : array[A..Z] of boolean);
{affiche les indices des éléments de t qui sont vrais, donc tous les faits acquis}
    var lettre : char;
begin write ('Faits acquis : '); for lettre := A to Z do if t[lettre] then write (lettre, ' ') end;

begin {program} entreprop (tf); entreeimplic (nr, tr); chainage (tf, nr, tr); sortie (tf) end.

```

**7-14° Dérivation des fonctions**, faire un programme de dérivation de fonctions d'une variable entrées sous forme préfixées. On utilisera une structure d'arbre pour les expressions, en se servant de champs différents pour les opérateurs à 0 place (les constantes), à 1 place (les fonctions) et à 2 places (les lois binaires).

```

program derivation;
type    tp = ^expression;
        expression = record op   : string [3]; {cas d'une variable ou constante}
                        case nbop : 0..2 of 1 : (arg : tp); {cas d'une fonction}
                                2 : (g, d : tp) {cas d'un opérateur binaire} end;

    var E, D : tp; ch : char;
procedure affiche (E : tp) ;
begin if E <> nil then case E^.nbop of
0 : write (E^.op); {"op" est ici une constante ou une variable}
1 : begin write (E^.op, '('); affiche (E^.arg); write (')') end;
2 : begin write('('); affiche (E^.g);write (')', E^.op, '('); affiche (E^.d); write (')') end end
end;
function oper2 (a : string) : boolean;
    begin oper2 := (a = '+') or (a = '-') or (a = '*') or (a = '/') end;
function oper1 (a : string) : boolean;
    begin oper1 := (a = 'sin') or (a = 'cos') or (a = 'ln') or (a = 'exp') end;
procedure entree (var E : tp); {demande une expression préfixée }
    var a : string [3];
    begin new (E); write ('Entrez le terme '); readln (a); E^.op := a;
    if oper2 (a) then begin E^.nbop := 2; write ('Premier opérande de ', a, ' ');
        entree (E^.g); write ('Second opérande de ', a, ' '); entree (E^.d) end
    else if oper1 (a) then begin E^.nbop := 1; write ('Argument de ', a, ' '); entree (E^.arg) end
    else E^.nbop := 0
    end;
function der (E : tp; v : char) : tp;
    var R : tp;
    begin new (R);
    if E^.op = 'sin' then begin R^.op := '*'; R^.nbop := 2;
        R^.g := der (E^.arg, v); R^.d^.op := 'cos'; R^.d^.nbop := 1; R^.d^.arg := E^.arg end
    else if E^.op = 'exp' then begin R^.op := '*'; R^.nbop := 2;
        R^.g := der (E^.arg, v); R^.d^.op := 'exp'; R^.d^.nbop := 1; R^.d^.arg := E^.arg end
    else if (E^.op = '+') or (E^.op = '-') then begin R^.op := E^.op;
        R^.nbop := 2; R^.g := der (E^.g, v); R^.d := der (E^.d, v) end
    else if E^.op = '*' then begin R^.op := '+'; R^.nbop := 2;
        R^.g^.op := '*'; R^.g^.nbop := 2; R^.g^.g := der (E^.g, v); R^.g^.d := E^.d ;
        R^.d^.op := '*'; R^.d^.nbop := 2; R^.d^.g := E^.g; R^.d^.d := der (E^.d, v) end
    else begin R^.nbop := 0; if E^.op = v then R^.op := '1' else R^.op := '0' end;
        der := R {dispose (R) est ici inutile, R est un pointeur local} end;
begin entree (E); affiche (E); write (' Dérivée '); new (D); D := der(E, 'x'); affiche (D); dispose (E) end.

```

**Remarque :** on continuera sur le modèle,

```

if E^.op = '/' then begin R^.op := '/'; R^.g^.op := '-'; R^.g^.nbop := 2;
with R^.g^.g^ do begin op := '*'; nbop := 2; g := der (E^.g, v); d := E^.d end;
with R^.g^.d^ do begin op := '*'; nbop := 2; g := E^.g; d := der (E^.d, v) end;
with R^.d^ do begin op := '*'; nbop := 2; g := E^.d; d := E^.d end

```

Exemple "3exp(x+2) - 5sin(x)" une fois entré est donné en sortie par :

```

((3) * (exp ((x) + (2)))) - ((5) * (sin (x))) et dérivé en :
(((0) * (exp ((x) + (2)))) + (3) * (((1) + (0)) * (exp ((x) + (2)))) - (((0) * (sin (x))) + ((5) * ((1)
* (cos (x)))))

```

**7-15°** Continuer le problème précédent par des fonctions de simplifications (calcul des termes dont les arguments sont numériques, retirer les parenthèses inutiles, appliquer des règles telles que  $0 = x = x$ ,  $1 * x = x$ , etc).

**7-16° Images réciproques d'une application quasi-affine** [Nehlig 92, Reveilles 91] Pour  $x, y$  entiers on définit l'application "quasi-affine" donnant  $x' = E[(ax + by + e) / w]$ , et  $y' = E[(cx + dy + f) / w]$  où  $a, b, c, d, e, f$  sont des entiers relatifs et  $w$  un entier positif non nul. L'image réciproque d'un point est l'intersection de deux "droites épaisses" et l'image réciproque d'ordre  $k$  est un certain domaine connexe limité par une courbe fractale. Ainsi pour 0, l'image réciproque d'ordre  $k$  est  $\{(u, v) / F^{(k)}(u, v) = (0, 0)\}$  pour les applications quasi-affines du type  $x' = [(ax + by) / (a^2 + b^2)]$ ,  $y' = [(ay - bx) / (a^2 + b^2)]$  on admet que :  $a / b = 1 / (q_1 + 1 / (1 + q_2))$  développée en fraction continue, ce contour s'exprimerait par un mot XYZXYZ de l'alphabet  $\{1, b, h, d, g\}$  symbolisant le mot vide, les déplacements bas, haut, droite et gauche.

Le contraire (qui est souligné) est involutif et défini par  $\underline{b} = h, \underline{d} = g$ , pour un chemin abcdef... c'est le reflet en miroir des contraires, soit le chemin ...fedcba.

$\phi_{q_1, q_2}$  est l'application définie pour trois mots par :

$$\phi_{q_1, q_2}(x, y, z) = (y, (z\underline{x})^{q_1}z(y(z\underline{x})^{q_1-1}z)^{q_2}, \underline{x}(z\underline{y}\underline{x})^{q_1})^{q_2+1})$$

Enfin si  $(X, Y, Z) = \phi_{q_1, q_2}^{(k)}(1, d, h)$  alors le contour est XYZXYZ ainsi pour  $k = 1$  ce contour est décrit par le mot "dhhdhghggbgbbdbdb".

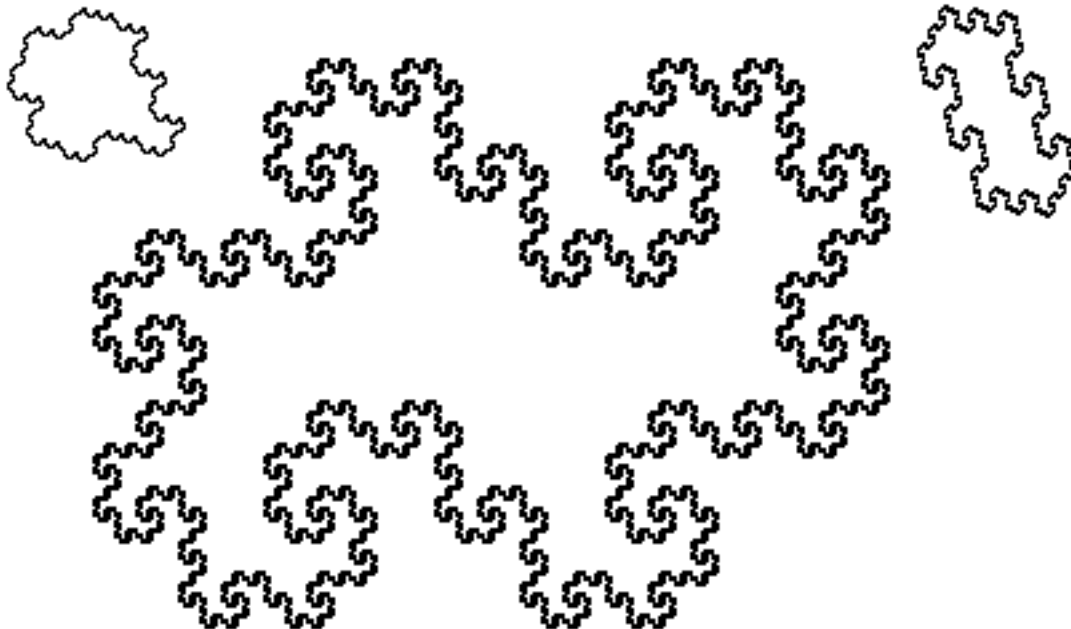
Programmer la fonction  $f$  et le tracé en affichant le mot en question.

(Voir les systèmes de Lindenmayer [Heudin 94]).

Réponse pour  $k = 3$  et  $q_1 = q_2 = 1$  (dessin de gauche)

hghgghghghhdhhdhg-ghghggbgbbghghggbgbbghghggbgbbghghggbgbbghghggbgbbghghgghghghhdhhdhg  
 gghghggbgbbghghggbgbbg-bgbbghghggbgbbghghggbgbbgbbdbdbbgbbdbdbdbdbbgbbghghggb  
 gbbghghggbgbbgbbdbdbbgbbdbdbdbdbbgbb-dbdbbgbbdbdbdbdb-dhhdhdhdbdbdhhdhdbdbdd  
 bdbbgbbdbdbdbdbdhhdhdhdbdbdhhdhdbdbdhhdhhdhdbdbdhhdhdbdbd-hhdhghgghghghhdh  
 hghghhdhhdhdbdbdhhdhdbdbdhhdhghgghghghhdhghghhdhhdhdbdbdhhdhdbdbdhhdh

Deux résultats pour  $q_1 = q_2 = 0$  et  $k = 7$  (à droite) puis  $k = 15$  (au centre).



```
program quasiaffine;
uses memtypes, quickdraw;
type      mot = ^doublet;
          doublet = record prem : char ; suivant : mot end;
```

```
procedure affiche (m : mot); {procédure de vérification}
begin if m <> nil then begin write (m^.prem); affiche (m^.suivant) end end;
```

```

function contraire (m, r : mot) : mot; {r sera le résultat, la fonction doit donc être appelée avec r = nil}
  var x : char; p : mot;
  begin if m = nil      then contraire := r
        else           begin new(p); x := m^.prem;
                        case x of 'b' : p^.prem := 'h'; 'h' : p^.prem := 'b';
                                'g' : p^.prem := 'd'; 'd' : p^.prem := 'g' end;
                        p^.suivant := r; contraire := contraire (m^.suivant, p) end
  end;

```

```

function concat (m, n : mot) : mot;
  var p : mot;
  begin if m = nil      then concat := n
        else           begin new (p) ; p^.prem := m^.prem;
                        p^.suivant := concat (m^.suivant, n); concat := p end
  end;

```

```

function puissance (x : mot ; n : integer) : mot;
  var p : mot;
  begin if n < 1      then puissance := nil
        else puissance:= concat (x, puissance (x, n-1))
  end;

```

```

procedure phi (q1, q2 : integer; var x, y, z : mot);
  var xc, a, b, c : mot;
  begin  new(xc); new(a); new(b); new(c); xc := contraire (x, nil);
        a := concat (z, xc);
        b := concat (puissance (a, q1 - 1), z);
        c := puissance (concat (contraire (y, nil), xc), q1);
        x := y; z := concat (xc, puissance (concat (a, c), q2 + 1));
        y := concat (b, puissance (concat (y, b), q2)); y := concat (a, y)
  end;

```

```

procedure parcours (var u, v : integer; m : mot);
{Le point courant étant (u, v), la procédure déplace ce point suivant les déplacements élémentaires figurant dans le mot m}
  var p : mot; x : char;
  begin if m <> nil      then begin x := m^.prem;
                              case x of 'b' :v:=v-1; 'h' :v:=v+1; 'd' :u:=u+1; 'g' :u:=u-1 end;
                              lineto (u, v);
                              parcours (u, v, m^.suivant) end
  end;

```

```

procedure dessin (q1, q2, k : integer);
  var i, u, v : integer; x, y, z, xc, yc, zc : mot;
  begin u := 150; v := 160; moveto(u, v); new(x); new(y); new(z);
        x := nil; y^.prem := 'd'; z^.prem := 'h'; y^.suivant := nil; z^.suivant := nil;
        if (q1 = 0) and (q2 = 0) then for i := 1 to k do  begin xc := contraire (x, nil);
                                                            x := y; y := z; z := concat (xc, concat (z, xc)) end
        else for i := 1 to k do phi(q1, q2, x, y, z);
        xc := contraire (x, nil); yc := contraire (y, nil); zc := contraire(z, nil);
        parcours (u, v, x); parcours (u, v, y); parcours (u, v, z);
        parcours (u, v, xc); parcours (u, v, yc); parcours (u, v, zc);
        affiche(x); write ('-'); affiche (y); write ('-'); affiche (z); write ('-'); affiche (xc) ; write ('-');
        affiche (yc); write ('-'); affiche (zc); {Ces affichages peuvent bien sûr être supprimés}
  end;

```

```

begin dessin (0, 0, 15) end.

```

**7-17° Solutions non isométriques du problème des reines de Gauss**

En cherchant les solutions sans considérer les 8 isométries du carré, on risque d'avoir plusieurs fois les mêmes, mais certaines sont globalement invariantes par une rotation. Plutôt que de vérifier à chaque fois si une solution a déjà été obtenue, il serait plus rapide de restreindre le champ de liberté de chaque reine.

On considère pour cela sur le damier  $n \times n$ , l'ensemble  $S_k$  des solutions telles que la dame la plus proche d'un coin soit en  $k$ -ième place (le coin étant la première). On a alors la partition  $S = S_1 + S_2 + S_3 + \dots + S_{E(n/2)}$ . Ces ensembles sont en effet disjoints et dans  $S_1$ , il n'y a que trois dames sur le pourtour. Si  $n$  est impair, une dame en  $E(n/2)+1$  d'un coin est en fait en  $E(n/2)$  du coin opposé.

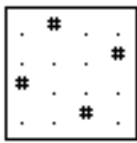
La première limitation à faire est donc, lorsque toutes les solutions de  $S_k$  auront été trouvées avec la dame de la première ligne en  $k$ , d'interdire les emplacements de distance  $\leq k$  à un coin.

Le champ de la première dame sera de 1 à  $E(n/2)$ .

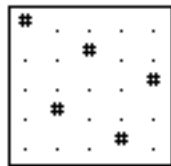
Cas de  $S_1$  : il faut remarquer que c'est seulement en ce cas que l'on peut avoir deux solutions symétriques par rapport à la diagonale, pour les éviter, dès qu'une solution de  $S_1$  est trouvée et si la position de la seconde dame est en  $k$ , on interdit la case symétrique : colonne 2 et ligne  $k$ .

Cas des rotations : pour  $k > 1$ , on peut avoir lorsqu'une solution est trouvée avec 2 ou 3 dames sur le pourtour en position  $k$ , on ne la prendra en compte que si la 4<sup>e</sup> l'est aussi.

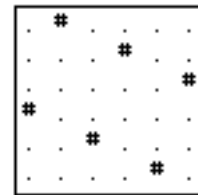
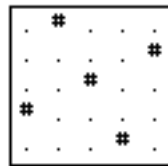
Pour le reste on introduit une liste chaînée des solutions de  $S_k$  en testant si une des trois rotations d'une solution s'y trouve déjà.



n = 4



n = 5



n = 6

```
program gauss;
```

```
const n = 8;
```

```
type table = array[1..n] of integer;
```

```
ptr = ^doublet;
```

```
doublet = record tab : table; suivant : ptr end;
```

```
var t : table; ES : set of 1..n; ch : char; SOL : ptr; v : integer;
```

```
function rotdroit (t1,t2 : table): boolean; {teste si t2 est la rotation d'un droit de t1}
```

```
var i : integer; non : boolean;
```

```
begin i := 1; non := false;
```

```
repeat non := (t2[t1[i]] <> n-i+1); i := i+1 until non or (i > n);
```

```
rotdroit := not (non) end;
```

```
function symcentre (t1, t2 : table): boolean; {test si 2 solutions sont symétriques / centre}
```

```
var i : integer; non : boolean;
```

```
begin i := 1; non := false;
```

```
repeat non := (t2[n-i+1] <> n-t1[i]+1); i := i+1 until non or (i > n);
```

```
symcentre := not (non) end;
```

```
procedure rajouter (t : table ; var liste : ptr ); {rajoute t en tête de la liste}
```

```
var i : integer; aux : ptr;
```

```
begin new (aux); aux^.suivant := liste;
```

```
for i := 1 to n do aux^.tab[i] := t[i];
```

```
liste:= aux end;
```

```
function present (t : table; liste : ptr): boolean;{teste si une rotation de t est dans file}
```

```
begin
```

```
if liste = nil then present := false
```

```
else if rotdroit (t, liste^.tab) then present := true
```

```
else if rotdroit (liste^.tab, t) then present := true
```

```
else if symcentre (t, liste^.tab) then present := true
```

```
else present := present (t, liste^.suivant) end;
```

```

function nonprise (i, j, k, l : integer) : boolean; {k<i ici}
  begin if j = l then nonprise := false
        else if abs (j-l) = k-i then nonprise := false else nonprise := true
  end;

```

```

function caselibre (l, c : integer) : boolean; {teste si en prise avec les lignes du dessus}
  var i : integer; drap : boolean;
  begin i := 1; drap := true;
  while drap and (i<l) do begin drap := nonprise (i, t[i], l, c); i := i+1 end;
  caselibre := drap
end;

```

```

procedure place (l, c, k : integer; var nb : integer); forward;

```

```

procedure reine (i, k : integer; var nb : integer);
  {tente de continuer à remplir t avec la dame i en t[i] déjà mise, k étant t[1]}
  var j : integer;
  begin
    if i=0 then for j := 1 to n div 2 do begin SOL := nil; place(1, j, j, nb) end
    else if i=n then if not (present (t, SOL)) then
      begin gotoxy (1, 20); nb := nb + 1;
        rajouter (t, SOL);
        if k = 1 then ES := ES + [t[2]];
        write ('solution n° ', nb, ' continuer ? '); ch := readchar;
        end
    else if i = n-1 then for j := k+1 to n-k+1 do place (n, j, k, nb)
    else if (i < k-1) or (i > n-k) then for j:=2 to n-1 do place (i+1, j, k, nb)
    else for j := 1 to n do place(i+1,j,k, nb)
  end;

```

```

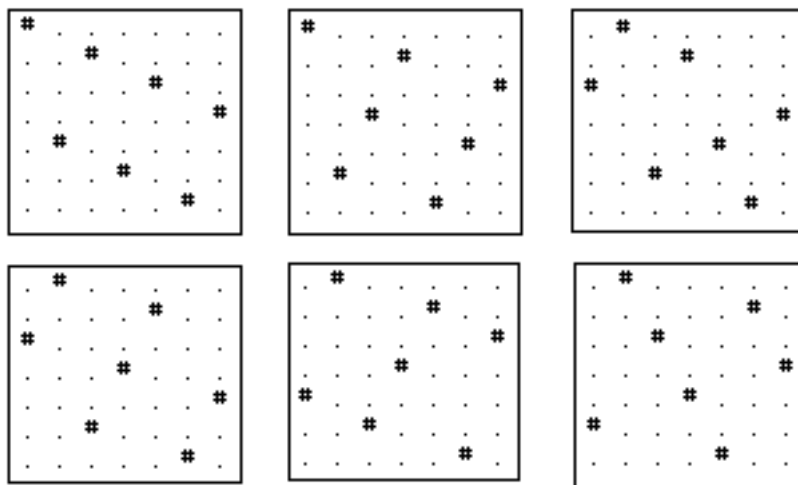
procedure place; {pour l, c, k affecte le tableau t et inscrit '#' sur l'écran}
  begin
    if caselibre(l, c) and ((k <> 1) or (c <> 2) or not (l in ES)) then
      begin t[l] := c; gotoxy (2*c, l); write ('#'); reine (l, k, nb); gotoxy (2*c, l); write ('.') end
  end;

```

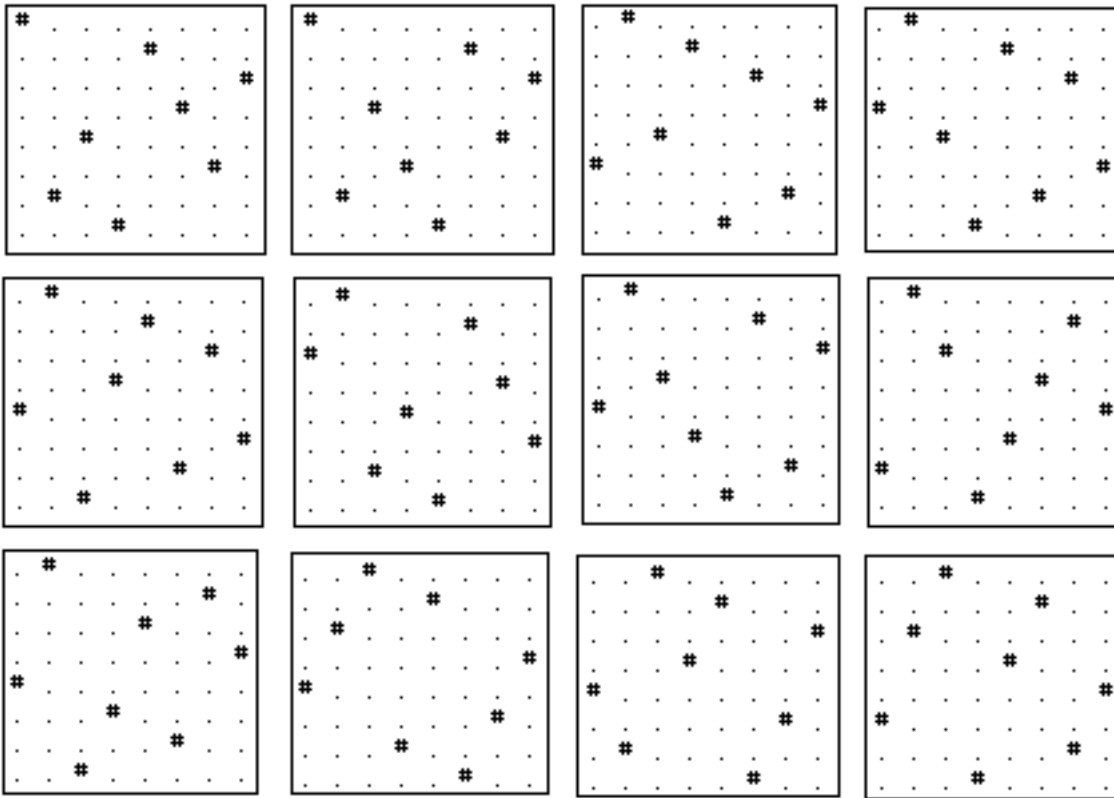
```

var l, c : integer; t1, t2 : table; { début du programme }
begin clearscreen; new (SOL); ES := []; v:=0;
  for l:=1 to n do for c := 1 to n do begin gotoxy(2*c, l); writeln('.') end;
  c := 0; reine (0, 0, c) end.

```



Les six solutions pour n = 7

Les douze solutions pour  $n = 8$ **7-18° Jeu du morpion**

Sur un damier  $n \times n$ , les noeuds sont occupés par -1 ou 1 (les deux joueurs) ou bien 0 (case vide), -1 représente les croix déjà placées par l'adversaire. Le but du jeu est d'aligner  $k$  pions à soi pour un  $k \leq n$  (on peut tester pour  $n = k = 3$ ). Le programme doit jouer contre l'utilisateur avec la stratégie suivante : à chaque étape où le programme doit jouer il faut vérifier les alignements.

S'il y a un  $(k-1)$ -alignement à soi, alors il faut le compléter, le jeu est gagné.

S'il y a un  $(k-1)$ -alignement de l'adversaire, il faut le contrer.

Sinon on va se placer dans la case correspondant à la croisée des alignements dont le total de ses pions et de ceux de l'adversaire est maximum. En fait il faudra pondérer les alignements en favorisant ceux qui sont presque achevés.

Représenter le tableau de tous les alignements possibles ( $2n + 2$  si  $k = n$ ), chacun d'entre eux représentant une case de départ, un sens (ligne, colonne, diagonale montante ou descendante), le nombre de cases à 1 déjà placé, à -1 et enfin un indicateur pour le fait ou cet alignement ne peut plus conduire à aucun succès pour les deux joueurs.

**7-19°** Faire une procédure de dessin d'un feuillage pour une profondeur donnée et une procédure récursive lisant un arbre binaire dont les fils gauche et droit sont formés chacun d'un facteur  $0 < k < 1$  et d'un virage  $0 < a < 45^\circ$  signifiant que si  $d$  est la longueur du rameau précédent, les deux suivants auront les longueurs réduites  $k_g d$  et  $k_d d$  déviés de  $a_g$  et  $a_d$  par rapport à la direction du rameau dont elles sont issues.

On peut commencer par écrire une procédure très simple où ces paramètres  $k_g$ ,  $k_d$ ,  $a_g$  et  $a_d$  sont toujours les mêmes, auquel cas il n'y a pas d'arbre à demander à l'utilisateur, seule la longueur  $d$  du premier rameau (le tronc) comptera.

**7-20° Lignes de niveaux**

Etant donnée une fonction réelle  $f$  de deux variables, on souhaite tracer  $k$  lignes de niveaux régulièrement répartis dans un intervalle  $[p, q]$  où l'on sait par un moyen quelconque que  $f$  prend ses valeurs sur un domaine donné  $[a, b] \times [c, d]$ . Le problème est de relier ces points de façon à former des lignes.

Chaque ligne correspondant à un fichier séquentiel dans lequel chaque nouveau point admis devra s'insérer suivant une stratégie à définir.

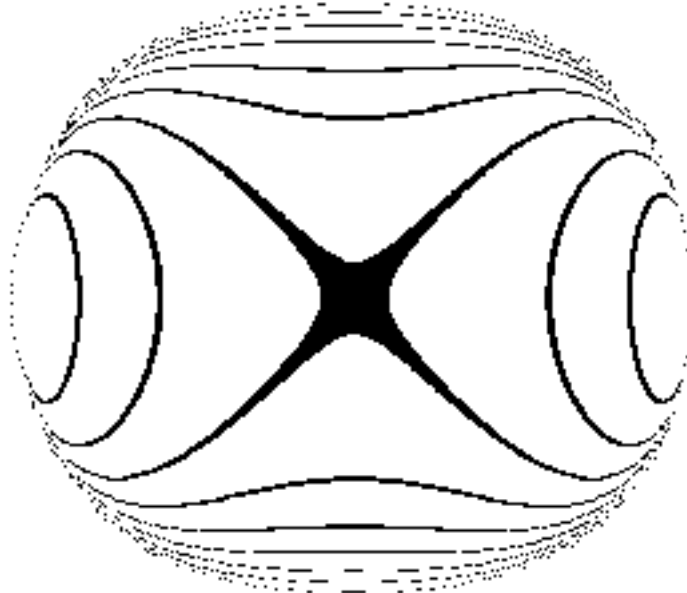
La procédure, une fois mise au point, sera surtout utilisée pour des fonctions n'ayant pas d'expression mathématique simple.

Le programme consiste simplement en un appel de lignes avec les arguments souhaités.

Exemple, les fenêtres de Viviani sont les intersections d'une sphère et d'un cylindre circulaire droit dont une génératrice est diamètre de la sphère. Leurs projections sur un plan tangent contenant ce diamètre sont :

$(1-x^2)^2 = \mu(1-x^2-y^2)$ , on pourra les avoir comme lignes de niveaux de la fonction :

$$(x, y) \in [-1, 1]^2 \rightarrow \mu = (1-x^2)^2 / (1-x^2-y^2)$$



```
uses memtypes, quickdraw; {uses crt, graph; en turbo 6}
const H = 270 ; L = 480;      {hauteur, largeur de l'écran}
A = -1.8; B = 2; C = -1; D = 1 ; {une fenêtre}
m = 20; {maximum de nombre de lignes de niveaux}

type
  ptr = ^triplet;
  triplet = record x, y : real; suivant : ptr end;
  var t : array [1..m] of ptr;
  {contient des files de points x, y de même valeur à epsilon près (on en veut k < m)}

procedure place (x, y : real); {place le point de vraies coord x,y}
begin xa:=x; ya:=y; moveto ( L*(x-A) div (B-A), H*(y-D) div (C-D)) end;

procedure trace(x, y :real); {trace le segment vers le point de vraies coord x,y}
begin lineto ( L*(x-A) div (B-A), H*(y-D) div (C-D)) end;

procedure point (x, y : real); begin place (x, y); trace (x, y) end;

function dis (r, s, t, u : real) : real; {donne la distance euclidienne entre (r, s) et (t, u)}
begin dis := abs (r-t) + abs (s-u) end;
```

```

procedure inserer (u : integer; i, j : real);
{insère le point i,j dans t[u] pour 0 < u < k, l'insertion doit se faire là où la somme des distances entre deux
points de la file est minimale y compris entre le premier et le dernier }
  var un, deux, nouveau : ptr; d0, d1 : real; k, p : integer;
begin new(nouveau); un:= t[u]; k := 1; p:= 0; {il faudra insérer après p}
nouveau^.x:= i; nouveau^.y:= j; nouveau^.suivant := nil;
if un <> nil then begin deux:= un^.suivant;
  if deux <> nil then begin d0 := dis (un^.x, un^.y, i, j) + dis (i, j, deux^.x, deux^.y);
    un := deux; deux := deux^.suivant; k := k+1 end;
  while deux <> nil do begin
    d1 := dis (un^.x, un^.y, i, j) + dis (i, j, deux^.x, deux^.y);
    if d1 < d0 then begin d0 := d1; p := k end;
    un := deux; deux := deux^.suivant; k := k+1 end ;
  if (k <> 1) and (dis (un^.x, un^.y, i, j) + dis (i, j, t[u]^x, t[u]^y) < d0) then p := k ;
  un := t[u]; deux := un^.suivant;
  for k := 1 to p-1 do begin un := deux; deux := deux^.suivant end;
  un^.suivant := nouveau; nouveau^.suivant := deux end
  else t[u] := nouveau end;

```

**procedure lignes** (n, k : integer; a, b, c, d, p, q, eps : real); {construit un tableau de k files dont chacune est formée par des points de  $[a, b] \times [c, d]$  correspondant à eps près aux k valeurs régulièrement réparties pour la fonction f dans [p, q]. Le domaine est parcouru suivant un quadrillage  $n \times n$ }

```

  var i, j, u : integer; x, y, v : real ; courant : ptr;
begin
place(a, c); trace(a, d) ; trace(b, d); trace(b, c); trace(a, c); place(a, 0); trace(b, 0); place(0, d); trace(0, c);
for i := 1 to n do for j := 1 to n do
  begin x := a + (b-a)*(i-1)/(n-1); y := c + (d-c)*(j-1)/(n-1); v := f(x, y);
  if (p <= v) and (v <= q) then begin
    u := round (1 + (v-p)*(k-1)/(q-p)); {est le rang éventuel de cette valeur}
    if abs (v-p-(q-p)*(u-1) / (k-1)) < eps then
      begin point (x, y); inserer (u, x, y) end
    end
  end;
end;
for u := 1 to k do if t[u] <> nil then
  begin courant := t[u]; place (t[u]^x, t[u]^y);
  while courant <> nil do begin trace( courant^.x, courant^.y); courant := courant^.suivant end
  end;
end;
{on appellera "lignes" avec les paramètres désirés}

```

Solution insérant les points suivant les abscisses croissantes :

```

procedure inserer (v, i, j : integer);
{insère le point i,j suivant les i croissants dans t[v] pour -m < v < m }
  var courant, pred, nouveau : ptr; fini : boolean;
begin new (nouveau); courant := t[v]; nouveau^.x := i; nouveau^.y := j; fini := false;
while (courant <> nil) and not (fini) do
  if courant^.x < i then begin pred := courant; courant := courant^.suivant end
  else fini := true;
nouveau^.suivant := courant;
if courant = t[v] then t[v] := nouveau
  else pred^.suivant := nouveau
end;

```

