

IREM

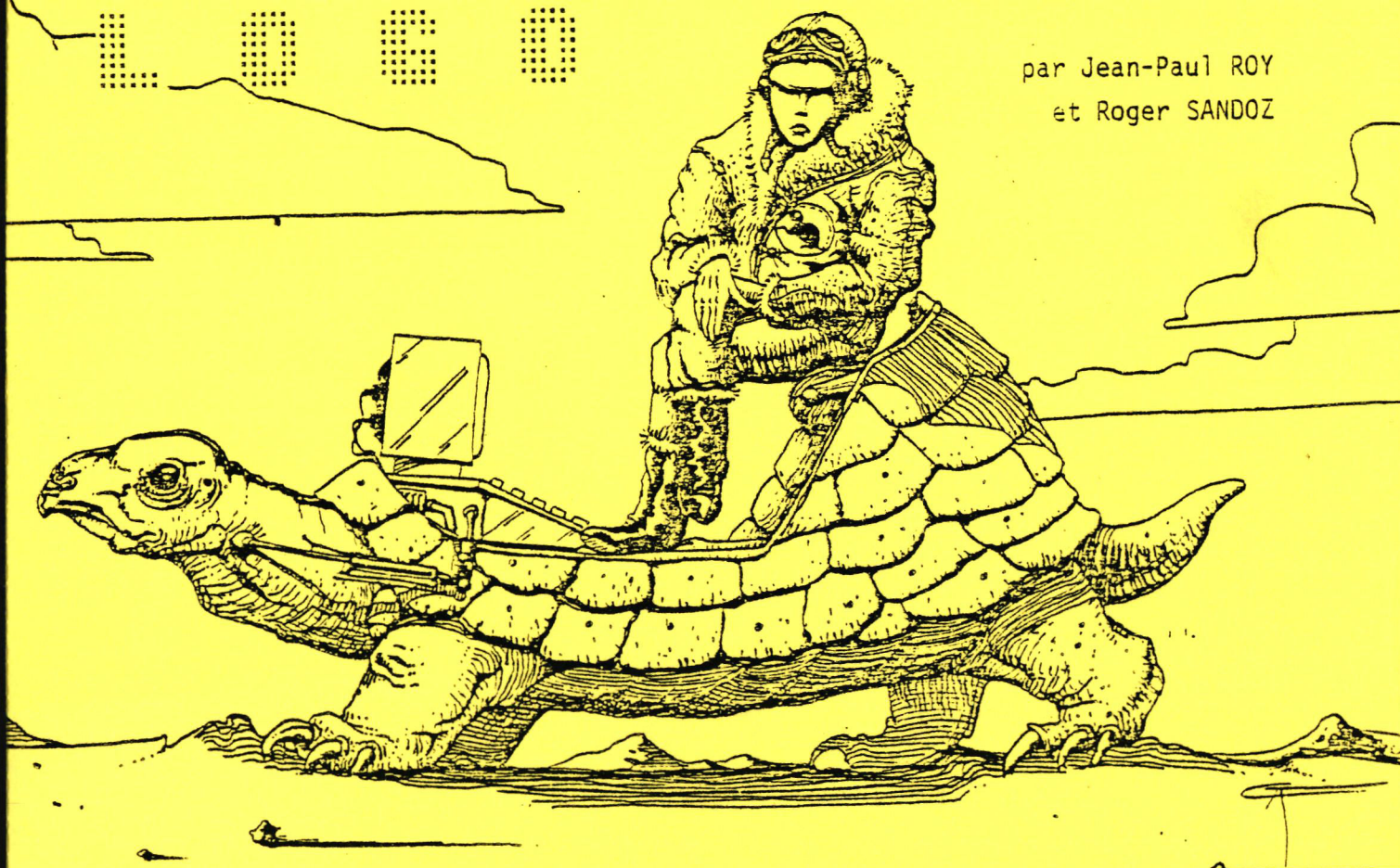
INSTITUT DE RECHERCHE SUR L'ENSEIGNEMENT DES MATHÉMATIQUES

47

juin 1983

MATHEMATIQUES
POUR
LOGO

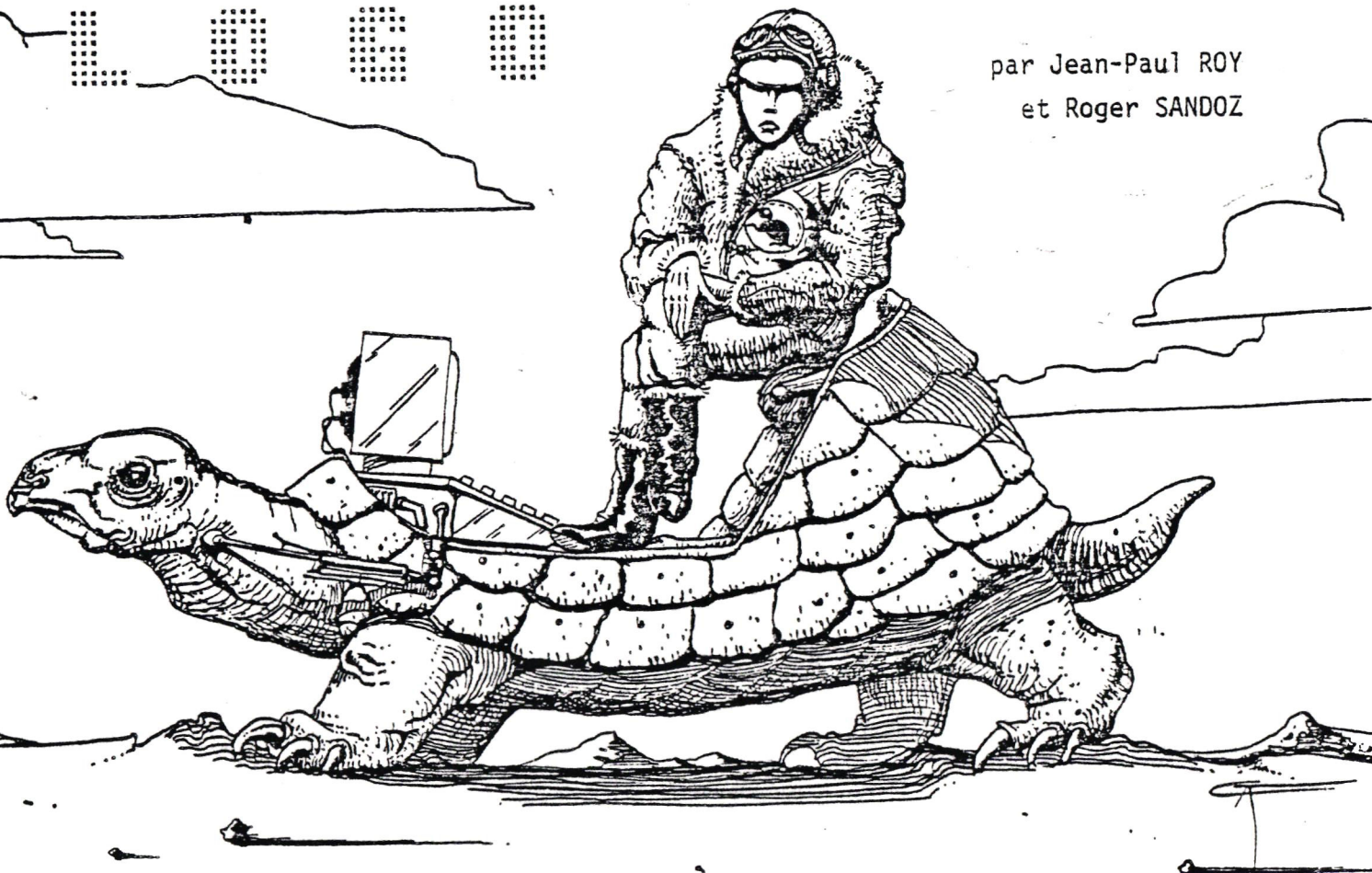
par Jean-Paul ROY
et Roger SANDOZ

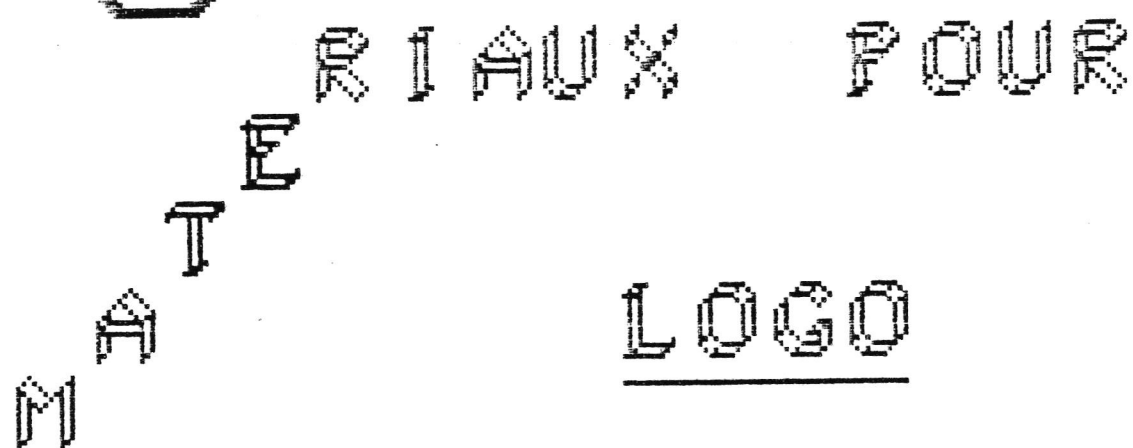


<u>NIVEAU</u>	Premier et second cycle
<u>PUBLIC</u>	Enseignants du 1er cycle / Enseignants et élèves du 2ème cycle
<u>SUJET</u>	Le langage LOGO Exemples d'applications
<u>OBJECTIF</u>	Aide à l'expérimentation

MOTERNAUX POLIQUES

par Jean-Paul ROY
et Roger SANDOZ





MATERIAUX POUR
LOGO

J. P. ROY ET R. SANDOZ
IREM DE PARIS-SUD

1982-83

SOMMAIRE

I. Introduction: du Monde de la Tortue à l'Univers des Listes.

- I.1 Avertissement.
- I.2 Repères bibliographiques.
- I.3 Les objets LOGO.
- I.4 Manipulation des Listes.
- I.5 Guidage de la Tortue.

II. Parcours en terrain récursif.

II.1 La récursivité: une méthode naturelle de programmation en LOGO.

II.2 Manipulations récursives de Listes.

- II.2.1 Ensembles.
- II.2.2 Arithmétique.
- II.2.3 Logique.
- II.2.4 Calcul symbolique.
- II.2.5 Jeux récursifs.
- II.2.6 Tris.
- II.2.7 Exercices et projets.

II.3 Guidage récursif de la Tortue.

- II.3.1 Polygones.
- II.3.2 Spirales.
- II.3.3 Arborescences.
- II.3.4 Courbes fractales.

III. LOGO: un langage "ouvert".

III.1 Generalites sur LISP. Comparaison avec LOGO.

III.2 LOGO en LOGO.

- III.2.1 Simulation de primitives.
- III.2.2 Modification de la boucle-système.

III.3 Dialogues.

IV. Appendices.

App 1 : Les "polynômes éparpillés" en LOGO.

App 2 : Tracés de courbes (polaire/cartésien/paramétrique).

App 3 : Une répartition géométrique des nombres premiers.

. DU MONDE DE LA
TORTUE
A L'UNIVERS DES
LISTES.

1. Nous avons voulu faire de ce document un ensemble d'exemples, aussi variés que possible (et en particulier ne se limitant pas au graphique) de programmes LOGO. Peu d'ouvrages (même américains) existant à l'heure actuelle sur le sujet, il nous a semblé utile de mettre à la disposition des enseignants des Écoles et Lycées un texte couvrant des sujets pouvant aller de la Logique aux Objets fractals, en passant par le traitement formel des polynômes ou des considérations plus théoriques sur les origines du langage LOGO.

Bref, les pages qui suivent peuvent être considérées au choix comme un support d'enseignement ou un aide à l'écriture de didacticiels. Cette dernière voie, peu explorée, est particulièrement prometteuse, au vu des capacités "tous azimuts" de ce langage, se traduisant par des facilités:

- sur le plan graphique (coordonnées polaires ou cartésiennes);
- au niveau du calcul arithmétique (entier ou flottant) ou ensembliste (par le biais des listes);
- pour le traitement des phrases et des mots (analyses de dialogues).

2. Ce que cet ouvrage n'aborde pas: le discours fréquent sur la légitimation épistémologique de la démarche LOGO comme outil d'apprentissage autonome. Bien que très pertinent dans des mains très avisées - ce qui n'est pas notre cas - cet aspect nous a paru marqué du sceau de la répétition, et a, bien involontairement, contribué à enfermer LOGO dans un rôle marginal de "langage pour les enfants".

3. La version utilisée dans ces pages est celle tournant sur APPLE II. Non que ce soit la meilleure, mais ce fut la seule disponible au début de ce travail, et nous avons évité le risque d'éventuelles modifications syntaxiques. De toutes façons, la "portabilité" de LOGO est telle que l'adaptation sur une autre machine ne devrait poser aucun problème.

4. Nos conventions d'édition. Elles sont de trois ordres:

- bien que n'existant pas réellement sur notre version, nous avons inclus dans le texte des programmes LOGO des "commentaires" débutant par un point-virgule. Ceci sous-entend que le reste de la ligne n'est pas pris en compte par l'interprète LOGO. Un moyen peu honorable d'y pallier pour une personne qui en voudrait de "vrais" est de définir la "procédure point-virgule" par:

```
? POUR ; :L
> FIN
? MONTRE [1 2 3] ; [CECI EST UN COMMENTAIRE]
[1 2 3]
?
```

- les graphiques: issue de "recopie d'écran graphique" sur imprimante une page graphique ne correspond pas toujours telle quelle au programme censé lui donner naissance. Pour deux raisons: d'abord des problèmes de cadrage à l'écran, qui auraient "noyés" l'essentiel d'une procédure dans une masse de détails dépendant trop de la machine, comme le nombre de points adressables, etc..., mais aussi

en second lieu parce que la plupart des dessins furent "travaillés" à l'aide d'un éditeur graphique spécialisé EZ-DRAW permettant entre autres de placer du texte sur la page graphique (normalement impossible sur l'APPLE), de choisir une police de caractères parmi plusieurs dizaines, etc...

- le texte lui-même: desolés pour les accents et les caractères français qui brillent par leur absence, mais notre imprimante n'acceptait pas à la fois ces derniers et les crochets LOGO, qui s'utilisent à profusion, et que nous avons privilégiés.

Notre version de LOGO ne nous a pas autorisé les minuscules.

5. Les thèmes discutés ne sont que très grossièrement "ordonnés" et chaque chapitre, voire paragraphe, est accessible isolément. Un corollaire en est que l'ordre des pages n'est pas celui d'une difficulté croissante.

Ce document n'est donc pas non plus un manuel d'initiation à LOGO, au sens où il enseignerait pas à pas la syntaxe du langage. Nous avons plutôt supposé acquis le vocabulaire de base (fixation des primitives utilisées en I.3-4-5) et insiste sur les mécanismes d'écriture de programmes LOGO, particulièrement le traitement récursif de listes ou de graphiques.

Des absentes, pourtant disponibles dans notre version: les "listes de propriété" ou p-listes. Nous avons observé que peu de versions les possédaient, aussi nous en avons fait l'économie.

6. Nos remerciements au Centre Mondial pour l'Informatique et les Ressources Humaines (et à ses animateurs) pour la possibilité d'utiliser les machines et la version de LOGO dont il est ici question.

Un dernier merci (et un coup de chapeau) aux élèves de 1ère de l'Ecole Alsacienne (Paris) qui se sont vus participer à la rédaction de ces notes dans le cadre de projets de programmation d'un cours optionnel d'Informatique normalement consacré au langage Pascal; plus précisément:

- Achille Beres (Jeux récursifs);
- David Fisher (Les Polynômes éparpillés).

Leur rédaction a été dans l'ensemble conservée en l'état, et témoigne de ce qu'il est possible de faire avec des élèves.

Enfin, la couverture est due à Guillaume Aretos, élève de la même Ecole, qui témoigne par une plume experte de sa vision du Monde plat de la Tortue A lui aussi, bravo...

7. Et bravo à tous ceux, des collèges ou Lycées, qui découvriront d'ici peu ce monde simple et extraordinaire qu'est celui de LOGO.

=====

I.2

- REPERES BIBLIOGRAPHIQUES -

- H. ABELSON et A. Di SESSA
"Turtle Geometry" [Une invitation a la Geometrie-Tortue]
(M.I.T. Press, 1980)
- G. BOSSUET [Une experience LOGO en CM2]
"L'ordinateur a l'école"
(P.U.F., 1982)
- S. PAPERT [Le manifeste LOGO]
"Jaillissement de l'esprit"
(Flammarion, 1981)
- Le rapport Schwartz [Des propositions sur l'Informatique dans
"Informatique et Education" les ecoles]
(La Documentation Francaise, 1981)

Articles de revues:

=====

"Some ideas on the educational use of computers"
(Harald Wertz, in ACM, Novembre 1981). [une experience francaise...]

MICRO-SYSTEMES:

- "LOGO: un langage d'avenir" (No 27, Janvier 1983).
- "Une Tortue en 3 dimensions" (No 29, Mars 1983).
- "LOGO et l'Intelligence Artificielle" (No 30, Avril 1983).

BYTE: Un numero consacre en LOGO (Aout 1982).

Rapports de recherche:

=====

Une quantite industrielle, centree sur les "Memos" des Universites du MIT (USA) et d'Edimbourg (Ecosse). On les trouve dans les bibliotheques universitaires; ils sont souvent axes sur les aspects educatifs et psychologiques de la demarche LOGO, par exemple chez des enfants en situation d'echec scolaire ou d'handicap physique. Certains developpent des themes particuliers, comme les programmes musicaux, l'apprentissage LOGO chez de tres jeunes enfants non encore alphabetises, la Tortue dans un champ gravitationnel, etc...

Le memoire suivant est d'un contenu particulierement riche:

"Comment transformer l'education en se servant de la technologie"
par S. Papert
(M.I.T. LOGO Memo No 8)

N.B. : Ce qui precede n'est qu'une selection : une bibliographie bien plus complete est fournie a la fin de l'ouvrage de Gerard BOSSUET.

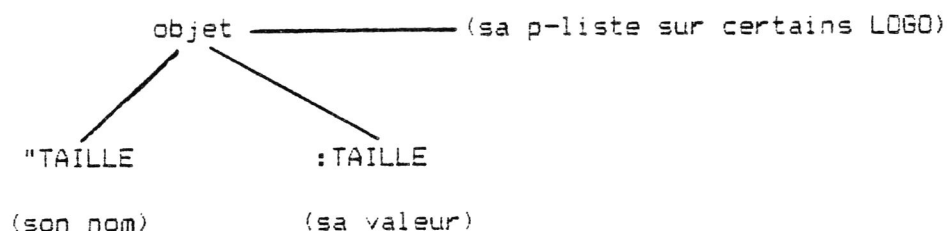
=====

I.3

- Les objets LOGO -

Les objets LOGO ne sont pas TYPES, au contraire des langages "classiques" comme BASIC ou LSE (un petit nombre de types de base, non extensibles) ou Pascal (l'utilisateur définit à volonté ses structures de données). Autrement dit, une valeur :X peut être un entier, un réel, un mot, une liste, etc...

Plus précisément, un "objet" LOGO possède la structure suivante:



Remarquez que l'on n'accède à un objet que par son nom ou sa valeur; si vous parlez à LOGO de l'objet X, l'interprète le traitera comme un appel à la procédure qui s'appelle X.

```

? DONNE "X 2 ; ces 3 lettres X n'ont...
? POUR X :X ; ...aucun...
> RT :X + 1 ; ...rapport !
> FIN
X EST DEFINI
? EC :X
2
? EC X
PAS ASSEZ D'INFO POUR X ; X est interprete comme une procedure.
? EC "X
X
? EC X :X ; les 2 lettres X n'ont aucune relation...
3
? EC :X
2

```

Les ACTEURS du petit monde LOGO:

1) Les "objets".

=====

a) les constantes.

```

- les "mots numeriques": 31
                        -31 (sans blanc)
                        31.416 ( -id- )

- les "mots litteraux" : "SALUT
                        "S
                        "!!SALUT!
                        "SALUT\ -LES\ -GARS (\ pour prendre - comme un
                        caractere ordinaire)

- les "listes" : [1 2 "HELLO [1 [] 2] 3] (5 elements)
                [[]] (1 element: la liste reduite a la
                liste vide)

```

b) les variables.

```

Une variable n'est pas "typee" et n'est accessible qu'a-travers son nom
"X" ou sa valeur :X (ou sa p-liste).
? DONNE "VAR1 [1 [2 3] 4]          ; se lit "donne a la variable dont le
?                                nom est VAR1 la valeur [1 [2 3] 4]
? ED :VAR1
1 [2 3] 4                        ; sans les crochets extérieurs...
? MONTRE :VAR1
[1 [2 3] 4]                      ; ...avec!
? DONNE "VAR2 :VAR1              ; donne a la variable dont le nom est
?                                VAR2 la valeur de celle dont le nom
?                                est VAR1.
? ED :VAR2
1 [2 3] 4
? DONNE "VAR3 "VAR2              ; donne a la variable dont le nom est
?                                VAR3 la valeur "VAR2.
? ED :VAR3
VAR2
? MONTRE CHOSE :VAR3
[1 [2 3] 4]                      ; (CHOSE :VAR3) = :VAR2 = :VAR1

```

L'affectation a donc pour syntaxe generale:

```
DONNE <nom> <valeur>
```

On pourra ainsi modifier la valeur d'une variable en écrivant par ex.:

```
DONNE "VAR <resultat de l'application d'une fonction a :VAR>
```

2) Les "Procedures".

Pour pouvoir travailler sur ces objets, il nous faut des "methodes de travail": les procedures vont tenir ce role; elles serviront d'interfaces entre l'utilisateur et l'interprete LOGO. Tout a ete fait - ou tente de l'etre - pour que le travail de l'eleve durant une session LOGO soit percu par lui comme une activite non directive: c'est l'eleve qui est cense apprendre a l'ordinateur COMMENT faire telle ou telle action, et ce de maniere tres interactive, le langage, interprete, se pretant bien a une simulation de dialogue.

S'il est vrai que le:

```
SYNTAX ERROR IN 2095
```

vaut souvent un:

```
NE SAIS PAS COMMENT CARRE
```

il n'en reste pas moins que le second message est moins frustrant que le premier, l'ordinateur semblant prendre l'erreur a son propre compte, tout en la localisant.

Sans compter que la localisation en question peut s'averer bien plus precise; comparez:

```
CARRE N'AIME PAS RECEVOIR [] COMME INFO
LIGNE 10 EN SORTANT DE TRACER
```

et:

```
DIVISION BY ZERO ERROR IN 1789
```

pouvant référencer une ligne comportant plusieurs instructions de calcul.

Ces "petits riens", qui concernent aussi par exemple l'incorporation d'un editeur de texte "pleine page", en l'occurrence une version d'EMACS, font partie de cette nouvelle exigence d'un bon "environnement de programmation".

La syntaxe generale d'une definition de procedure est:

```
POUR <nom de procedure> <liste d'arguments>
  <corps de la procedure>
FIN
```

dans laquelle:

<nom de procedure> est un mot litteral non quote;
 <liste d'arguments> est une liste de valeurs de parametres; ces parametres sont "locaux" a la procedure.
 <corps de la procedure> est une suite d'appels de procedures.

Exemple:

```
? POUR DIRE :FOIS :TEXTE
> REPETE :FOIS [EC :TEXTE DONNE "TEXTE SP :TEXTE]
> FIN
DIRE EST DEFINI
? DIRE 2
PAS ASSEZ D'INFO POUR DIRE ; deux parametres attendus !
? DONNE "ESSAI [MAIS BONJOUR LOGO]
? DIRE 2 :ESSAI
MAIS BONJOUR LOGO
BONJOUR LOGO
? MONTRE :ESSAI
[MAIS BONJOUR LOGO] ; phenomene tres important: le parametre
? ; ESSAI a ete "passe par valeur"; cela
? ; signifie que malgre les affectations
? ; qu'il a subies en cours de procedure, il
? ; ressort intact avec la valeur qu'il avait
? ; a l'entree de la procedure. Si l'on veut
? ; qu'il demeure modifie, ce n'est pas sa
? ; valeur (non referencee) qu'il faut passer
? ; en parametre, mais son "nom", comme dans
? ; l'exemple suivant:
? POUR DIREBIS :FOIS :TEXTE
> REPETE :FOIS [ECRIE CHOSE :TEXTE DONNE :TEXTE SAUF PREMIER CHOSE :TEXTE]
> FIN
DIREBIS EST DEFINI
? DIREBIS 2 "ESSAI
BONJOUR LOGO
LOGO
? MONTRE :ESSAI
[LOGO] ; on dit que la procedure DIREBIS vient
; d'effectuer un "effet de bord" sur la
; valeur de son parametre "ESSAI.
```

En LOGO, la notion de "fonction" est cachee dans celle de procedure: une fonction est une procedure qui RETOURNE une valeur:

```
? POUR CARDINAL :LISTE
> SI VIDEP :LISTE [RT 0] ; sous-entendu: [RT 0 STOP]
> RT 1 + CARDINAL SP :LISTE
> FIN
CARDINAL EST DEFINI
? CARDINAL [LES CAROTTES SONT CUITES]
NE SAIT QUE FAIRE AVEC 4 ; sorry, my dear !
```

Notez que l'on sort de la fonction des qu'un "RETOURNE" est rencontre, autrement dit, l'ordre STOP, cense stopper la procedure en cours pour ramener a la procedure appellante, est ici inutile.

La fonction precedente est "recursive", i.e. elle s'appelle elle-meme pour evaluer sa valeur. La RECURSIVITE est plus qu'un atout dans l'optique de la programmation LOGO, c'est un veritable style (cf. le chapitre "Parcours en terrain recursif").

Des complements sur la procedure POUR seront fournis en III.1.1.

=====

I.4

- MANIPULATION DES LISTES -

Voir II.3 pour les distinctions entre "objets", "listes" et "mots" LOGO.

Primitive =====	Parametre(s) =====	Description =====
LISTEP	un objet	retourne "VRAI si l'objet est une liste.
MOTP	"	" (...est un mot).
NOMBREP	"	" (...est un nombre).
VIDEP	un mot ou une liste	retourne "VRAI si le parametre est le mot vide " ou la liste vide [].
EGALP	deux objets	teste l'egalite.
MEMBREP	un objet et une liste	teste l'appartenance au premier niveau de l'objet a la liste.
PREMIER	un mot ou une liste	retourne la 1ere lettre du mot ou le premier element de la liste.
DERNIER	" "	" (...dernier...).
SAUFPREMIER SP	" "	retourne le mot prive de sa premiere lettre ou la liste privée de son premier element.
SAUFDERNIER SD	" "	" (...dernier...).
ITEM	un nombre et une liste	ITEM :N :L retourne le N-ieme element de la liste L.
COMPTE	une liste	retourne le nombre d'elements de la liste.
MOT	des mots (*)	retourne le mot constituant la concatenation des mots.
PH	des objets (*)	retourne la liste constituee des objets mis bout a bout: PH [....] [****] ---> [....****]
LISTE	" (*)	retourne la liste dont les elements sont les objets: LISTE obj1 obj2 ---> [obj1 obj2]
ASCII	caractere	en retourne le code ASCII.
CAR	nombre 0..255	retourne le caractere dont il est le code ASCII.

====
N.B. Une paire de () autorise en principe un nombre quelconque d'arguments,
==== mais ceci depend de l'implementation. Voir par ex. les tests de la page
ci-contre.

Une batterie d'exemples:

```
? EC LISTEP "HELLO
FAUX
? EC MOTP "HELLO
VRAI
? EC NOMBREP "HELLO
FAUX
? EC LISTEP [T H X 1 1 3 8]
VRAI
?
? EC VIDEF [] ; la liste vide.
VRAI
? EC VIDEF [[]]
FAUX ; un seul element: la liste vide.
? EC VIDEF " ; le mot vide.
VRAI
? EC MEMBREP "TORTUE [LA TORTUE DANS LE MONDE LOGO]
VRAI
? EC MEMBREP 3 [1 2 [3] 4]
FAUX ; appartenance "au premier niveau".
? MONTRE PREMIER [[1 2] 3 4]
[1 2]
? EC PREMIER [[1 2] 3 4]
1 2
? MONTRE SP [[1 2] 3 4]
[3 4]
? MONTRE DERNIER [[1 2] 3 4]
4
? MONTRE SD [[1 2] 3 4]
[[1 2] 3]
? MONTRE DERNIER SD SP PREMIER [[1 2 3 4] 5 6] ; et une aspirine, une...
3
? EC ITEM 3 [LA TORTUE DANS LE MONDE LOGO]
DANS
? EC COMPTE [LA TORTUE DANS LE MONDE LOGO]
6
? EC MOT "LO "GO
LOGO
? EC MOT "LO " ; " est le mot vide.
LO
? MONTRE PH "LO "GO
[LO GO]
? MONTRE (PH [1 2] [2] [3 4])
[1 2 2 3 4] ; fusion.
? MONTRE LISTE 1 2
[1 2]
? MONTRE (LISTE [1 2] [2] [3 4])
[[1 2] [2] [3 4]] ; enveloppement.
? EC ASCII "E
69
? EC CAR 69 ; CAR peut etre utilise pour envoyer des caracteres
E ; speciaux (de controle) a certains peripheriques, par
; exemple CAR 17 dans notre cas pour une recopie d'ecran
; graphique sur imprimante.
```

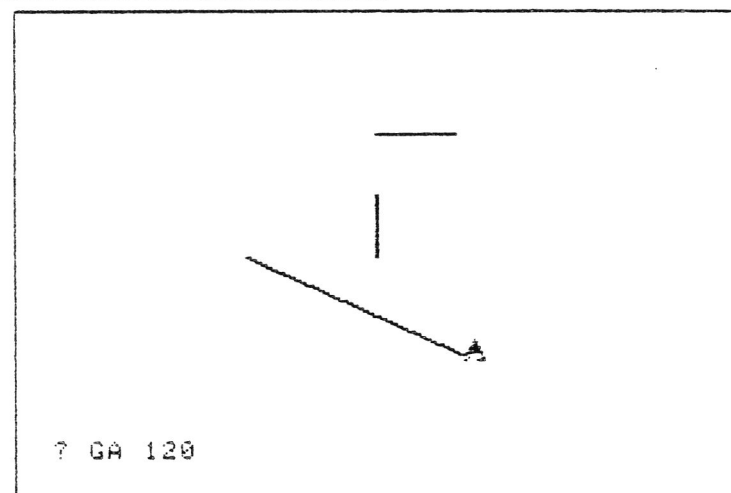
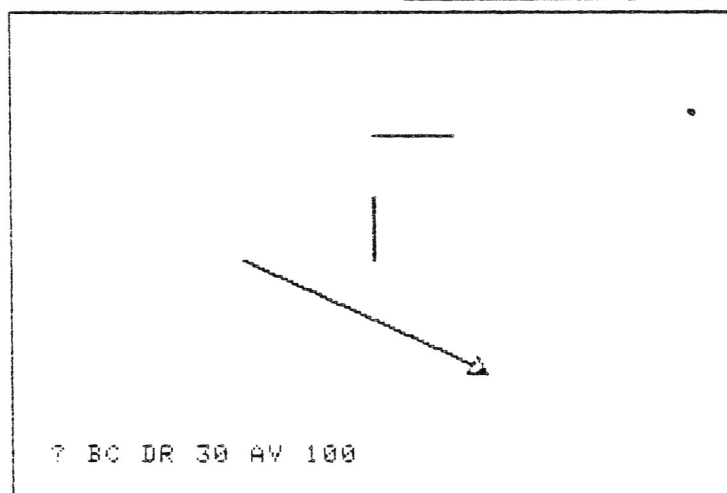
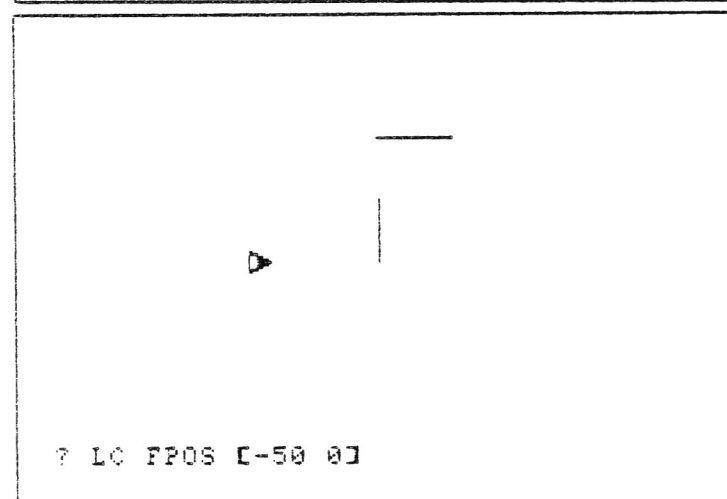
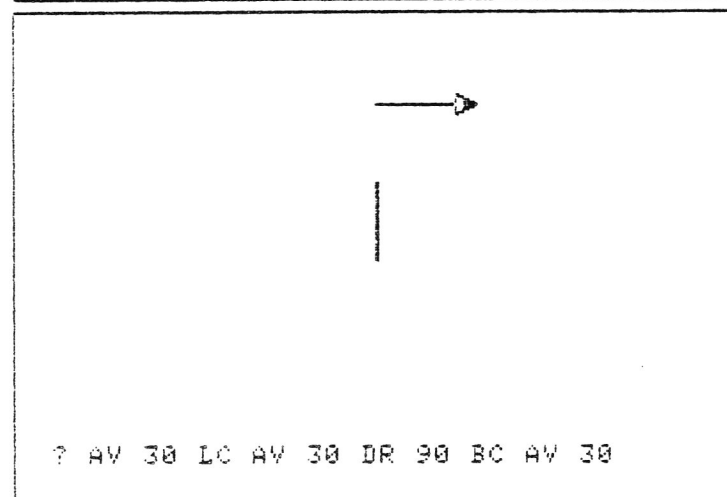
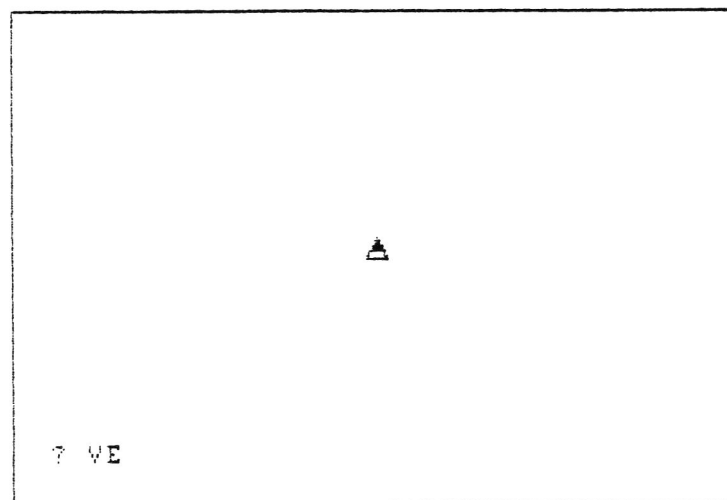
I.5

- GUIDAGE DE LA TORTUE -

Il est donc ici question des primitives graphiques, limitées à celles qui seront utilisées plus loin. La Tortue est un point sur l'écran, dessiné en général comme un petit triangle pointant vers une direction donnée (le "cap") et ayant un "crayon" levé ou non à la base du triangle.

Primitive =====	Parametre(s) =====	Description =====
VIDEcran VE		vide l'écran graphique et place la Tortue au centre, cap 0. Le contenu de l'éditeur est perdu.
MONTREtortue MT		affiche la Tortue.
CACHETortue CT		cache la Tortue.
LEVEcrayon LC		la Tortue se déplace sans dessiner.
BAISSEcrayon BC		retour au dessin.
GOMMEcrayon		la gomme.
INVERSEcrayon		dessine avec la couleur complémentaire du point sur lequel est la Tortue.
POINT	une liste de deux nombres	laisse un point à l'emplacement dont les coordonnées sont fournies par la liste.
POS		retourne une liste de deux nombres donnant les coordonnées de la Tortue.
FPOS	une liste de deux nombres	déplace la Tortue sur le point dont les coordonnées sont fournies par la liste.
AVANCE AV	un nombre	déplace la Tortue d'une longueur égale au nombre, sans modifier le cap.
RECULE RE	"	" (en sens opposé).
DROITE DR	"	effectue une rotation à droite de la Tortue, l'angle de rotation en degrés étant fourni par le nombre.
GAUCHE GA	"	" (à gauche).

=====



• PARCOURS
EN
TERRAIN RECURSIF •

"La meilleure façon de marcher,	POUR MARCHER
C'est encore la notre,	METTRE.UN.PIED.DVANT.LAUTRE
C'est de mettre un pied d'avant l'autre,	MARCHER
Et d recommencer..."	FIN

II.1 - La RECURSIVITE: une methode naturelle de programmation en LOGO -

La structure de "liste", qui est à la base de LOGO, et plus encore les moyens d'accès à une liste - PREMIER et SAUFPREMIER - imposent souvent un traitement sur une liste entière en fonction des traitements séparés sur le premier élément, puis sur le reste de la liste. Ceci différencie totalement ce style de programmation avec ceux qui sont imposés par les langages plus classiques comme FORTRAN, BASIC ou LSE manipulant plutôt la structure de "tableau" (les "pointeurs" de Pascal permettent par contre de simuler fort efficacement les listes et plus généralement toutes les structures arborescentes).

le "style LOGO"

le "style FORTRAN"

Pour traiter une liste L:
Si L = [] : traitement special
Sinon - traiter le 1er element
 - traiter le reste de la liste

Pour traiter un tableau T(N):
Pour I=1 à N:
 - traiter T(I)

(longueur de liste arbitraire)

(longueur de tableau fixe)

Il n'est pas possible en LOGO (1) d'accéder directement (i.e. en une seule indirection) au n-ième élément d'une liste; on doit pour cela parcourir tous les précédents, et d'ailleurs une liste ne véhicule pas d'information sur le nombre de ses éléments. Voici par exemple cette fonction LONGUEUR:

```
? POUR LONGUEUR :L          ; COMPTE dans notre version.
> SI VIDE? :L [RT 0] [RT 1 + LONGUEUR SP :L]
> FIN                        ; l'appel recursif.
LONGUEUR EST DEFINI         ; on comptabilise le 1er element.
? LONGUEUR []
0
? LONGUEUR [[EXEMPLE DE] BELLE FONCTION RECURSIVE]
4
```

La construction d'un objet (une fonction par exemple) est dite "RECURSIVE" si elle définit l'objet en question en termes de cet objet lui-même, ou d'un objet "de la même classe".

Nous insistons beaucoup sur les aspects "construction" et "definition" qui sont à la base de la recursivité. Ce que l'on construit recursivement peut être un cercle, une suite de nombres, une procédure LOGO ou un raisonnement mathématique. Une telle construction recursive est plus proche d'une description que d'un algorithme proprement dit, et une programmation recursive est quasiment naturelle dès lors qu'elle porte sur un objet dont la construction elle-même est DÉJÀ recursive !

(1) : notre version possède ITEM, mais le parcours de la liste n'en est pas moins séquentiel, même s'il est écrit en langage machine.

```

? POUR FAC :N
> SI :N = 0 [RT 1] [RT :N * FAC :N - 1]
> FIN
FAC EST DEFINI
? EC FAC 3
6

```

Essayons de "tracer" (i.e. suivre pas a pas) le déroulement des calculs engagés lors de l'appel FAC 3:

FAC 3	FAC 3 = 3 * 2 = 6
demande	donc
FAC 2	FAC 2 = 2 * 1 = 2
demande	donc
FAC 1	FAC 1 = 1 * 1 = 1
demande	donc
FAC 0	----> 1

Cet exemple - simple et classique - montre bien le mécanisme "en 2 temps" d'un appel récursif: la descente vers le "point d'arrêt" de la recursion puis la remontée vers l'appel lui-même, les résultats intermédiaires sous le bras.

Nous allons développer tout au long de ce polycopie un "arsenal" de procédures récursives, en espérant vous convaincre au bout du compte que, contrairement à l'opinion souvent admise - par les personnes ayant été formées au FORTRAN par exemple - la récursivité est une méthode élégante, compréhensible et simple en programmation. Le prix à payer pour ces avantages peut être le temps de calcul, mais de deux choses l'une: ou la récursivité est évidemment inutile pour un algorithme donné, et alors on s'en passe (pensez au calcul de $C(n,p) = C(n-1,p) + C(n-1,p-1)$ qui est lent), ou bien son élégance cache une quasi-impossibilité de s'en passer, sauf à faire de l'algorithmique de haut vol (pensez aux Tours de Hanoi ou au Baguenaudier, cf II.2.5).

Pour en finir avec ces généralités, ne développez pas le "syndrome FAC" lorsque vous pensez à la récursivité: cette dernière peut cacher des phénomènes plus complexes (appels récursifs successifs, appels co-récursifs,...). Enfin, il y a les "fausses récursivités" (comme celle de MARCHER, au début du chapitre), pour lesquelles l'appel récursif est dit "terminal" et qui ne font que cacher une brave iteration. Un point important en ce qui concerne la plupart des interprètes LOGO: ils savent détecter ces "faux" appels récursifs et les transforment en boucle "Tant que". Ceci différencie - aussi - LOGO des compilateurs de langages récursifs (Pascal) qui n'ont pas la même attitude:

Interprète LOGO

```

POUR MARCHER
MARCHER
FIN

```

(tourne sans arrêt)

Compilateur Pascal

```

Procédure MARCHER;
Begin
  MARCHER
End;

```

(est stoppé après quelques temps
par débordement de la "pile")

Exercices : voir II.2.7

II.2.1 Ensembles.

=====

Ce chapitre, outre qu'il va enrichir LOGO de nouvelles procédures (intersection, réunion et différence symétrique) va nous permettre de manier des listes, des fonctions avec passage des paramètres, tout cela en terrain récursif, le seul que LOGO affectionne....

Nous allons commencer par une fonction typique du LOGO ; elle aura pour but de nettoyer les listes des éléments répétés plusieurs fois:

```
? POUR NET :L
> SI VIDE? :L [RT :L]
> SI MEMBRES PREMIER :L SP :L [RT NET SP :L]
> RT METF PREMIER :L NET SP :L
> FIN
NET EST DEFINI
?
? MONTRE NET [1 3 5 2 3 1 7] ;suppression a gauche
[5 2 3 1 7]
```

Comment fonctionne donc cette procédure ?

Si le premier élément de la liste L figure dans la liste restante, alors on le laisse de côté et on nettoie la liste restante. Sinon, on garde le premier élément de la liste et on nettoie le reste de la liste.

On va reiterer ce procédé jusqu'à obtenir la liste vide et ainsi donc on retournera une liste d'éléments distincts.

Nous voici maintenant armés pour comprendre les deux procédures qui vont nous permettre de déterminer la différence symétrique de deux ensembles:

```
? POUR DIFSYM :A :B ; on suppose :A et :B nettoyées...
> RT PH DIF :A :B DIF :B :A ; la définition: (A-B) U (B-A)
> FIN
DIFSYM EST DEFINI
?
? POUR DIF :A :B ; les éléments de :A non dans :B
> SI VIDE? :A [RT :A]
> SI MEMBRES PREMIER :A :B [RT DIF SP :A :B]
> RT METF PREMIER :A DIF SP :A :B
> FIN
DIF EST DEFINI
?
? MONTRE DIF [3 2 4 -2 5 7] [2 8 5 0 1 7 4]
[3 -2]
? MONTRE DIFSYM [3 2 4 -2 5 7] [2 8 5 0 1 7 4]
[3 -2 8 0 1]
```

Sur notre lancée nous allons voir comment moyennant une petite astuce on peut définir les opérations réunion et intersection en utilisant qu'une seule procédure pour remplir les listes réunion et intersection

```
? POUR INTER :A :B
> LOCAL "J
> DONNE "J 1 ; 1 = code de l'intersection.
> RT PARCOURS :A []
> FIN
INTER EST DEFINI
?
? MONTRE INTER [1 3 5 7 8] [2 5 3 7 0]
[3 5 7]
```

```

? POUR REUNION :A :B
> LOCAL "J
> DONNE "J 2 ; 2 = code de la reunion.
> RT PARCOURS :A :B
> FIN
REUNION EST DEFINI
?
? MONTRE REUNION [2 4 5 7] [ 4 3 6 5 ]
[4 3 6 5 2 7]
?
? POUR PARCOURS :A :LIS ; on regarde si chaque element de A est ou
> SI VIDEP :A [RT :LIS] ; n'est pas dans B
> DONNE "E PREMIER :A
> SI OU ET MEMBREP :E :B :J = 1 NON OU MEMBREP :E :B :J = 1
[RT PARCOURS SP :A METD :E :LIS]
> RT PARCOURS SP :A :LIS
> FIN
PARCOURS EST DEFINI

```

La conditionnelle dans la procedure PARCOURS s'annonce en clair ainsi:

SI (E~~E~~B ET J=1) OU (E~~E~~B ET J#1) alors ajouter E dans :LIS.

Un examen attentif de cette procedure montre qu'elle remplit son contrat aussi bien pour l'intersection (J=1) que pour la reunion (J#1), etant entendu que la valeur de LIS n'est pas la meme dans les deux cas.

On peut evidemment determiner la reunion de deux ensembles sans utiliser la procedure PARCOURS, en procedant par exemple de la maniere suivante:

```

? POUR REUNION :L1 :L2
> RT NET UNION :L1 :L2
> FIN
REUNION EST DEFINI

? POUR UNION :L1 :L2
> SI VIDEP :L1 [RT :L2]
> SI MEMBREP PREMIER :L1 :L2 [RT UNION SP :L1 :L2]
> RT METF PREMIER :L1 UNION SP :L1 :L2
> FIN
UNION EST DEFINI

```

La logique de la procedure UNION est la meme que celle de la procedure NET; voir II.2.7 (exercice 10).

=====

II.2.2 Arithmétique.

=====

A present examinons le comportement de LOGO en arithmétique.

Une illustration tres classique de la recursivite est la recherche du PGCD de deux nombres en utilisant l'algorithme de Monsieur Euclide:

```
? POUR PGCD :A :B ; :A et :B sont des naturels non nuls.
> SI :A < :B [RT PGCD :B :A]
> SI RESTE :A :B = 0 [RT :B] [RT PGCD :B RESTE :A :B]
> FIN
PGCD EST DEFINI
?
? EC PGCD 12 30
6
```

On notera l'utilisation de RT qui renvoie au niveau superieur et que son omission plongerait LOGO en plein desarroi , ce que traduit le cri de detresse "NE SAIT QUE FAIRE AVEC 36 AU MOMENT DE QUITTER PGCD"...

On peut, toujours recursivement, trouver le PPCM de deux nombres:

"On cherche le plus petit multiple du plus grand qui soit aussi un multiple du plus petit".

```
? POUR PPCM :A :B
> SI :B > :A [RT PPCM :B :A]
> RT PPCM1 :A :B 1 ; 1 = le 1er multiple de :A
> FIN
PPCM EST DEFINI
?
? POUR PPCM1 :A :B :N
> SI RESTE :A*:N :B=0 [RT :A*:N]
> RT PPCM1 :A :B :N+1
> FIN
PPCM1 EST DEFINI
?
? EC PPCM 12 40
120
```

Il est bien entendu possible d'utiliser le PGCD deja defini:

```
? POUR PPCM.FACILE :A :B
> RT :A * :B / PGCD :A :B
> FIN
PPCM.FACILE EST DEFINI
```

Et maintenant divisons pour mieux regner...

Les trois modestes procedures suivantes vont nous permettre de trouver tous les diviseurs d'un entier naturel.

```
? POUR DIVIS,:A
> (LOCAL "DIV "COMPL)
> DONNE "DIV [] ; la liste des diviseurs de A inferieurs a
> ; la racine carree de A.
> DONNE "COMPL [] ; la liste des diviseurs "complementaires".
> DIVISEUR :A 1
> TEST DERNIER :DIV=PREMIER :COMPL
> SIV [DONNE "COMPL SP :COMPL ]
> RT PH :DIV :COMPL
> FIN
DIVIS EST DEFINI
```

```

? POUR DIVISEUR :A :B          ; generer les diviseurs de A a-partir de B,
> SI :B * :B > :A [STOP]      ; et inferieurs a la racine carree de A.
> SI RESTE :A :B = 0 [AJOUTE :B]
> DIVISEUR :A :B+1
> FIN
DIVISEUR EST DEFINI
?
? POUR AJOUTE :B
> DONNE "DIV METD :B :DIV
> DONNE "COMPL METF QUOTIENT :A :B :COMPL
> FIN
AJOUTE EST DEFINI
?
? MONTRE DIVIS 18
[1 2 3 6 9 18]

```

Quel est donc le procede utilise?

On teste tous les nombres inferieurs ou egaux a la racine carree de A en remarquant que la connaissance d'un diviseur X nous donne en prime le divi-diviseur A/X. C'est ainsi qu'on remplira simultanement les listes DIV et COMPL.

Il nous faudra recoller avec precaution ces deux listes pour que dans le bouquet final nous ne trouvions pas deux fois le meme diviseur.

On ne saurait parler d'arithmetique sans dire un mot des nombres premiers. Commencons par determiner la suite des nombres premiers inferieurs a un nombre donne superieur a 1.

```

? POUR SUITE :X
> LOCAL "PREM
> SI :X = 2 [RT [2]]
> DONNE "PREM [2 3]
> RT RECHERCHE 1
> FIN
SUITE EST DEFINI
?
? POUR RECHERCHE :N          ; genere la suite des 6n 1
> LOCAL "Y
> DONNE "Y 6*:N-1
> SI :Y>:X [RT :PREM]
> ENRICHIR :Y :PREM
> SI :Y+2>:X [RT :PREM]
> ENRICHIR :Y+2 :PREM
> RT RECHERCHE :N+1
> FIN
RECHERCHE EST DEFINI
?
? POUR ENRICHIR :Z :A          ; mettre ou non Z dans PREM.
> LOCAL "RG                    ; RG = rang de A dans PREM
> DONNE "RG PREMIER :A
> SI RESTE :Z :RG=0 [STOP]
> SI OU :Z<:RG * :RG :RG=DERNIER :PREM [DONNE "PREM METD :Z :PREM STOP]
> ENRICHIR :Z SP :A
> FIN
ENRICHIR EST DEFINI
?
? MONTRE SUITE 40
[2 3 5 7 11 13 17 19 23 29 31 37]

```

Par quels procedes diaboliques ces trois procedures nous permettent-elles d'arriver a nos fins?

Voyons cela de plus pres.

La procedure SUITE nous rappelle que le programme se refuse a traiter des cas triviaux et nous renvoie a nos doigts...

La procedure RECHERCHE sait que les "premiers" ≥ 5 ne se cachent que sous les nombres de la forme $6n-1$ ou $6n+1$ et, par consequent, elle ne fera appel a la procedure ENRICHIR que pour ces nombres a la forme bien singuliere, en la dispensant d'une vaine recherche (divisibilite par 2 et 3).

Ainsi donc la procedure ENRICHIR cherche dans la liste PREM deja creee un diviseur premier au nombre Z qu'on lui propose. Apres une recherche infructueuse jusqu'a la racine ou lorsqu'elle n'a plus de munitions (aucun element de PREM n'a reussi a diviser Z), elle baisse les bras et enrichit PREM d'une nouvelle recrue.

Etre premier, cela se merite...

Puisque les nombres premiers refusent obstinement tout diviseur autre qu'eux-memes et un, interessons-nous a des nombres plus cooperatifs, qui acceptent volontiers une decomposition en facteurs premiers plus interessante:

```
? POUR FACTEURS :X
> LOCAL "DEC
> DONNE "DEC []
> RECD1 :X 2
> RT :DEC
> FIN
FACTEURS EST DEFINI
?
? POUR RECD1 :X :N
> SI RESTE :X :N=0 [DONNE "DEC METD :N :DEC RECD1 QUOTIENT :X :N :N STOP]
> SI :N=3 [RECD2 :X 1 STOP]
> RECD1 :X :N+1
> FIN
RECD1 EST DEFINI
?
? POUR RECD2 :X :N
> SI RESTE :X 6*:N-1=0 [DONNE "DEC METD 6*:N-1 :DEC RECD2 QUOTIENT :X
6*:N-1 :N STOP]
> SI RESTE :X 6*:N+1=0 [DONNE "DEC METD 6*:N+1 :DEC RECD2 QUOTIENT :X
6*:N+1 :N STOP]
> SI :X=1 [STOP]
> SI NON (6*:N+1)*(6*:N+1) < :X [DONNE "DEC METD :X :DEC STOP]
> RECD2 :X :N+1
> FIN
RECD2 EST DEFINI
?
? EC FACTEURS 120
2 2 2 3 5
```

Passons sur la procedure FACTEURS qui se contente d'appeler les autres procedures pour remplir DEC. Voyons la procedure RECD1 qui n'a d'autre ambition que de s'occuper des diviseurs 2 et 3, car ceux-ci n'acceptent pas les formes $6n-1$ et $6n+1$.

Apres avoir rempli son contrat la procedure RECD1 passe la main a la procedure RECD2 qui devra terminer le travail commence.

Tout d'abord remarquons que RECD2 cherchera tous les diviseurs parmi les nombres de la forme $6n-1$ et $6n+1$. On est ainsi sûr d'atteindre tous les diviseurs premiers et même rien qu'eux, car si $6n-1$ par exemple n'est pas premier il admet au moins deux diviseurs premiers qui lui sont inférieurs or ces deux diviseurs auront déjà été examinés. Il en résulte qu'en s'arrêtant à temps, soit quand $X=1$ ou lorsque $6n:N+1 > \text{RAC} : X$ nous obtiendrons tout simplement la décomposition en facteurs premiers de X .

On notera l'usage très fréquent qui est fait de la récursivité (5 fois) qui allège considérablement la programmation en LOGO.

=====

II.2.3 Logique.

=====

L'exemple que nous allons développer est une application typique des possibilités de traitement symbolique de LOGO. Il s'agit d'écrire une procédure affirmant qu'une formule logique donnée est ou non une tautologie (i.e. prend la valeur "vrai" pour toutes valeurs de vérité des propositions P, Q, R,... qui la composent.

Nous n'utiliserons pas les tables de vérité, fort lourdes à manipuler, le temps d'établir une telle table est exponentiel en le nombre de propositions qui la composent, et il faut de plus engendrer les permutations des valeurs de vérité.

Nous nous bornerons à la "Logique des Propositions", un sous-langage de la Logique du 1er ordre, qui manipule des "propositions" notées P, Q, R,... ou tout ce que vous voudrez (symbolique...) qui ne soit pas un des connecteurs qui seront utilisés, à savoir:

# (le non)	> (l'implication)	ET	OU
------------	-------------------	----	----

Libre à vous d'augmenter le vocabulaire (par exemple par l'équivalence ou le "ou-exclusif") et de modifier les règles de transformation en conséquence.

Plus précisément, la "grammaire" de la Logique des Propositions est la suivante:

```

FBF ::= LITTERAL / [ CONNECTEUR FBF FBF ] / [# FBF] ; recursif !
LITTERAL ::= PROPOSITION / [ # PROPOSITION ]
PROPOSITION ::= P / Q / R / S / ...
CONNECTEUR ::= ET / OU / >

```

Le mode de représentation choisi est le mode préfixe, usuel en LOGO, mais rien ne vous empêche d'écrire une fonction de conversion infixe <--> préfixe.

Le principe ici utilise est la "mise sous forme de clauses", qui est une suite de transformations à appliquer à une FBF ("formule bien formée"), pour produire une FBF qui est une "conjonction de disjonctions de littéraux" (sic)

En clair, il y a 3 règles de "chirurgie" sur la liste représentant la FBF de départ:

1) Eliminer les implications:

Il s'agit de remplacer [$>$ FBF1 FBF2] par:

[OU [# FBF1] FBF2]

2) Réduire le champ des négations:

On applique la règle [# [# FBF]] $\rightarrow$ FBF, ainsi que les règles de De Morgan:

```

[# [OU FBF1 FBF2]]  $\rightarrow$  [ET [# FBF1] [# FBF2]]
[# [ET FBF1 FBF2]]  $\rightarrow$  [OU [# FBF1] [# FBF2]]

```

3) Distribuer les OU sur les ET:

i.e. appliquer les règles de distributivité usuelles:

```

[OU [ET FBF1 FBF2] FBF3]  $\rightarrow$  [ET [OU FBF1 FBF3] [OU FBF2 FBF3]]
[OU FBF1 [ET FBF2 FBF3]]  $\rightarrow$  [ET [OU FBF1 FBF2] [OU FBF1 FBF3]]

```

On peut alors montrer que cette série de "re-ecritures" preserve la "validité", i.e. le fait d'être ou non une tautologie. La formule bien formée qui est obtenue est dite "sous forme de clauses", une clause étant une suite finie de disjonctions de littéraux. La formule est alors une tautologie si et seulement si chaque clause en est une (à cause de la table de vérité du ET) !

Le programme qui suit va donc mettre la formule proposée sous la forme de clauses (les fonctions R1, R2 et R3) et construire la liste (ENS) des clauses (la fonction VIDER); cette liste va subir deux transformations:

- chaque élément (clause) va se voir amputer des OU inutiles (on revient à un OU n-aire). La fonction BALAYAGE s'en charge;
- chaque clause tautologique est éliminée (on vérifie si elle contient une paire P et [# P]). C'est le but de la fonction REDUCTION.

La formule est donc une tautologie si et seulement si l'ensemble résultant des clauses est vide. Si ce n'est pas le cas, le programme donne la liste des clauses le composant. On peut considérer ces clauses comme des "sous-buts" à atteindre pour prouver le théorème, dans le cas où les valeurs de vérité ne sont pas connues. Mais ceci commence à sentir fortement l'Intelligence Artificielle et le lecteur intéressé est invité à consulter (la liste est loin d'être exhaustive...):

- "Principles of Artificial Intelligence"
par Nils N. NILSSON
Addison-Wesley, 1982
- "Mathematical Theory of Computation"
par Z. MANNA
McGraw Hill, 1974

Ces deux auteurs vont bien sur considérablement plus loin; par exemple, pour le sujet qui nous intéresse, ils étendent ces résultats à la Logique quantifiée du 1er ordre, où des complications liées aux variables quantifiées surgissent, qui demandent un traitement spécial (skolemisation).

Quelques fonctions d'intérêt public:

```
? POUR 1ER :L ; pas de contrainte sur la première lettre...
> RT PREMIER :L ; ...d'un identificateur en LOGO.
> FIN
1ER EST DEFINI
?
? POUR 2ND :L
> RT PREMIER SP :L
> FIN
2ND EST DEFINI
?
? POUR 3IEME :L
> RT PREMIER SP SP :L
> FIN
3IEME EST DEFINI
?
? POUR ATOMIQUE :FBF
> RT OU VIDEF :FBF MOTF :FBF
> FIN
ATOMIQUE EST DEFINI
?
?
?
?
?
?
```

```

?      ; La boucle-système du programme:
?
? POUR LOGIQUE
> EC [- - - - -]
> EC [VOTRE FORMULE BIEN FORMEE:]      ; on ne vérifie PAS la syntaxe !!!
> CLAUSES PREMIER LL                  ; LL = liste lue au clavier. On en...
> LOGIQUE                             ; ...prend le 1er élément afin de...
> FIN                                  ; ...l'entrer avec les crochets.
LOGIQUE EST DEFINI
?
? POUR CLAUSES :FBF
> DONNE "FBF R1 :FBF                  ; pour...
> MONTRE :FBF                         ; ...voir...
> DONNE "FBF R2 :FBF                  ; ...les...
> MONTRE :FBF                         ; ...résultats...
> DONNE "FBF R3 :FBF                  ; ...intermédiaires.
> MONTRE :FBF
> LOCAL "ENS DONNE "ENS []            ; la liste des clauses, à remplir.
> VIDER :FBF                          ; dans ENS.
> EC [LISTE DES CLAUSES:] MONTRE :ENS ; pas très joli.
> DONNE "ENS REDUCTION BALAYAGE :ENS ; un coup de peigne !
> EC [LA MEME APRES REDUCTION:] MONTRE :ENS ; mieux.
> SI VIDE :ENS [EC [> > > > > FORMULE TOUJOURS VRAIE !...] STOP] ; cqfd!
> EC [CETTE FORMULE ADMET POUR CLAUSES :]
> LISTER :ENS
> FIN
CLAUSES EST DEFINI
?
?      ; les 3 fonctions de transformation:
?
? POUR R1 :FBF
> SI ATOMIQUE :FBF [RT :FBF]
> SI EGALP 1ER :FBF "> [RT METF "OU LISTE LISTE "# R1 2ND :FBF R1 3IEME
:FBF]
> TEST EGALP 1ER :FBF "#
> SIVRAI [RT METF "# LISTE R1 2ND :FBF]
> RT METF 1ER :FBF LISTE R1 2ND :FBF R1 3IEME :FBF
> FIN
R1 EST DEFINI
?
? POUR R2 :FBF
> SI ATOMIQUE :FBF [RT :FBF]
> TEST EGALP 1ER :FBF "#
> SIFAUZ [RT METF 1ER :FBF LISTE R2 2ND :FBF R2 3IEME :FBF]
> TEST LISTEP 2ND :FBF
> SIFAUZ [RT :FBF]
> SI EGALP 1ER 2ND :FBF "# [RT R2 2ND 2ND :FBF]
> SI EGALP 1ER 2ND :FBF "OU [RT R2 METF "ET LISTE METF "# (LISTE R2 2ND
2ND :FBF) METF "# LISTE R2 3IEME 2ND :FBF]
> SI EGALP 1ER 2ND :FBF "ET [RT R2 METF "OU LISTE METF "# (LISTE R2 2ND
2ND :FBF) METF "# LISTE R2 3IEME 2ND :FBF]
> FIN
R2 EST DEFINI
?
? POUR R3 :FBF
> SI NON MEMBRE "ET :FBF [RT :FBF]
> TEST EGALP 1ER :FBF "OU
> SIFAUZ [RT (LISTE "ET R3 2ND :FBF R3 3IEME :FBF)]
> SI ET LISTEP 2ND :FBF EGALP 1ER 2ND :FBF "ET [RT (LISTE "ET R3 (LISTE
"OU R3 2ND 2ND :FBF R3 3IEME :FBF) R3 (LISTE "OU R3 3IEME 2ND :FBF R3
3IEME :FBF))]

```

```

> SI ET LISTEP 3IEME :FBF EGALP 1ER 3IEME :FBF "ET [RT (LISTE "ET R3
  (LISTE "OU R3 2ND :FBF R3 2ND 3IEME :FBF) R3 (LISTE "OU R3 2ND :FBF
  R3 3IEME 3IEME :FBF))]
> RT R3 (LISTE "OU R3 2ND :FBF R3 3IEME :FBF)
> FIN
R3 EST DEFINI
?
?      ; le traitement de la liste des clauses:
?
? POUR VIDER :FBF      ; et remplir :ENS.
> SI MOTP :FBF [DONNE "ENS METF :FBF :ENS STOP]
>      ; un ET separant deux clauses, on les vide separement:
> SI EGALP 1ER :FBF "ET [VIDER 2ND :FBF VIDER 3IEME :FBF STOP]
>      ; sinon c'est une negation elementaire:
> DONNE "ENS METF :FBF :ENS
> FIN
VIDER EST DEFINI
?
? POUR BALAYAGE :L
> SI VIDEP :L [RT :L]
> LOCAL "PREM DONNE "PREM BALAI 1ER :L      ; coup de balai sur le 1er.
> SI VIDEP :PREM [RT BALAYAGE SP :L]      ; on elimine la clause [].
> RT METF :PREM BALAYAGE SP :L      ; ...et on continue.
> FIN
BALAYAGE EST DEFINI
?
? POUR BALAI :C      ; sur une seule clause.
> LOCAL "C1 DONNE "C1 []
> FILTRER :C      ; et met le resultat dans :C1
> RT :C1
> FIN
BALAI EST DEFINI
?
? POUR FILTRER :C
>      ; filtrer une clause et retenir les OU.
> SI VIDEP :C [STOP]
> SI ET LISTEP :C EGALP 1ER :C "OU [FILTRER 2ND :C FILTRER 3IEME :C STOP]
> DONNE "C1 METF :C :C1
> FIN
FILTRER EST DEFINI
?
? POUR REDUCTION :ENS
>      ; elimination des clauses tautologiques.
> SI VIDEP :ENS [RT :ENS]
> SI TAUTOLOGIE 1ER :ENS [RT REDUCTION SP :ENS]
> RT METF 1ER :ENS REDUCTION SP :ENS
> FIN
REDUCTION EST DEFINI
?
? POUR TAUTOLOGIE :CLAUSE
> SI VIDEP SP :CLAUSE [RT "FAUX]      ; clause reduite a un litteral.
> TEST ATOMIQUE 1ER :CLAUSE
> SIVRAI [SI MEMBREP (LISTE "# 1ER :CLAUSE) SP :CLAUSE [RT "VRAI] [RT
  TAUTOLOGIE SP :CLAUSE]]      ; la definition...
> SI MEMBREP 2ND 1ER :CLAUSE SP :CLAUSE [RT "VRAI] ; ...d'une tautologie.
> RT TAUTOLOGIE SP :CLAUSE      ; le 1er element est OK; on passe
> FIN      ; au suivant.
TAUTOLOGIE EST DEFINI
?
? POUR LISTER :L
> EC [- - - - -]
> DEBITER :L
> EC [- - - - -]
> FIN
LISTER EST DEFINI

```

```

? POUR DEBITER :L
> SI VIDEP :L [STOP]
> EC 1ER :L
> DEBITER SP :L
> FIN
DEBITER EST DEFINI ; ouf...

```

Des exemples d'exécution:

```

? LOGIQUE
- - - - -
VOTRE FORMULE BIEN FORMEE:
[> P Q]
[OU [# P] Q] ; apres R1,
[OU [# P] Q] ; apres R2,
[OU [# P] Q] ; apres R3.
LISTE DES CLAUSES:
[[OU [# P] Q]] ; une seule clause.
LA MEME APRES REDUCTION:
[[Q [# P]]] ; elimination du OU.
CETTE FORMULE ADMET POUR CLAUSES:
- - - - -
Q [# P] ; Q ou (non P)...
- - - - -
VOTRE FORMULE BIEN FORMEE:
[> P P]
[OU [# P] P]
[OU [# P] P]
[OU [# P] P]
LISTE DES CLAUSES:
[[OU [# P] P]] ; une seule clause...
LA MEME APRES REDUCTION:
[] ; ...tautologique....
[] ; ...donc supprimee !
> > > > FORMULE TOUJOURS VRAIE !... ; formule valide.
- - - - -
VOTRE FORMULE BIEN FORMEE: ; un peu plus serieux:
[> [ET [> P Q][> Q R]] [> P R]] ; la transitivite de ==>
; apres R1:
[OU [# [ET [OU [# P] Q] [OU [# Q] R]]] [OU [# P] R]]
; apres R2:
[OU [OU [ET P [# Q]] [ET Q [# R]]] [OU [# P] R]]
; apres R3:
[ET [ET [OU [OU P Q] [OU [# P] R]] [OU [OU P [# R]] [OU [# P] R]]] [ET
[OU [OU [# Q] Q] [OU [# P] R]] [OU [OU [# Q] [# R]] [OU [# P] R]]]] ; sic
LISTE DES CLAUSES:
[[OU [OU [# Q] [# R]] [OU [# P] R]] [OU [OU [# Q] Q] [OU [# P] R]] [OU
[OU P [# R]] [OU [# P] R]] [OU [OU P Q] [OU [# P] R]]]
LA MEME APRES REDUCTION:
[]
> > > > FORMULE TOUJOURS VRAIE !...
- - - - -
VOTRE FORMULE BIEN FORMEE: ; i.e.  $P \Rightarrow (Q \Rightarrow P)$ 
[> P [> Q P]]
[OU [# P] [OU [# Q] P]]
[OU [# P] [OU [# Q] P]]
[OU [# P] [OU [# Q] P]]
LISTE DES CLAUSES:
[[OU [# P] [OU [# Q] P]]]
LA MEME APRES REDUCTION:
[]
> > > > FORMULE TOUJOURS VRAIE !... ; tautologie bien connue...
- - - - -

```

VOTRE FORMULE BIEN FORMEE:

```
[> [> P Q] [ET P Q]]
[OU [# [OU [# P] Q]] [ET P Q]]
[OU [ET P [# Q]] [ET P Q]]
[ET [ET [OU P P] [OU P Q]] [ET [OU [# Q] P] [OU [# Q] Q]]]
```

LISTE DES CLAUSES:

```
[[OU [# Q] Q] [OU [# Q] P] [OU P Q] [OU P P]]
```

LA MEME APRES REDUCTION:

```
[[P [# Q]] [Q P] [P P]]
```

CETTE FORMULE ADMET POUR CLAUSES:

```
- - - - -
P [# Q]
Q P
P P
- - - - -
```

Ce dernier exemple pour montrer que le programme n'est pas parfait, et demanderait quelques "coups de peigne" supplémentaires, importants si l'on veut poursuivre dans l'optique de la résolution de problèmes.

N'oubliez pas que la mise sous forme de clauses, entre les lignes pointillées précédentes, signifie que:

- deux lignes sont séparées par des ET
- sur chaque ligne, les littéraux sont séparés par des OU.

On voit alors immédiatement les extensions à apporter au programme : ne pas écrire deux fois le même littéral sur une même ligne, et "factoriser" par tout littéral se trouvant sur chaque ligne, et isolé sur l'une des lignes.

En d'autres termes, il aurait été préférable d'obtenir les clauses suivantes:

```
- - - - -
P [# Q]
Q P
P
- - - - -
```

puis: P tout seul !

Ceci démontre que $[(P \Rightarrow Q) \Rightarrow (P \text{ et } Q)] \Leftrightarrow P$

Plus généralement, un programme destiné à "résoudre" un problème (i.e. à "construire" une démonstration d'un théorème), va mettre la formule proposée sous forme de clauses:

```
-----
A11 A12 ...A1n
A21 A22 ...A2m
. . . . .
. . . . . i.e. ET ( Ai1 OU Ai2 OU ... )
. . . . . i=1
Ak1 Ak2 ...Akp
-----
```

Démontrer le théorème revient alors à démontrer chaque disjonction:
Ai1 OU Ai2 OU...

ce qui se fera en tâchant de montrer que Ai1 est vrai. Si oui, c'en est fini pour cette clause, et on passe à la suivante; si non, on essaye avec Ai2, et ainsi de suite. Si on est bloqué sur une clause, on remonte au dernier choix effectuée, et l'on essaye le suivant (programmation par "back-track"; le lecteur intéressé peut étudier le programme des 8 dames donné en II.2.5, ou s'initier au langage PROLOG - PROGRAMMATION en LOGique - qui n'est autre qu'un immense démonstrateur de théorèmes, très sophistiqué. D'ailleurs, l'effort japonais sur les ordinateurs de la 5ème génération portera pour une bonne part sur l'implémentation de ce type de langage...).

II.2.4 Calcul Symbolique.

Nous allons donner l'une des méthodes pour déterminer la somme, le produit et la dérivée de polynômes en X . Le but ici n'étant pas d'avoir une méthode performante, mais d'illustrer sur un exemple le calcul symbolique.

Commençons par transformer l'écriture d'un polynôme en X en une liste représentant la suite de ses coefficients.

```

? POUR FORME :P
> RT FORME [] :P
> FIN
FORME EST DEFINI
?
? POUR FORME :P1 :P
> LOCAL "M
1 > SI VIDE P :P [RT COMPL :P1 :I]
2 > SI PREMIER :P = "+" [DONNE "M PREMIER SP :P] ; le + est inutile...
3 > SI PREMIER :P = "-" [DONNE "M MOT PREMIER :P PREMIER SP :P] ; ...pas le -
4 > SI NON MEMBREP PREMIER :P [+ -] [DONNE "M PREMIER :P DONNE "P METF "+" :P
5 > SI VIDE P1 [DONNE "P1 METD COEF :M :P1 DONNE "I DEGRE :M RT FORME :P1
  SP SP :P]
6 > SI :I < DEGRE :M [EC [LE POLYNOME N'EST PAS ECRIT SUIVANT LES PUISSANCES
  DECROISSANTES ] RENVOIE "NIVEAUSUP]
7 > DONNE "P1 COMPL1 :P1 :I-1-DEGRE :M
8 > DONNE "P1 METD COEF :M :P1
9 > DONNE "I DEGRE :M
10 > RT FORME :P1 SP SP :P
> FIN
FORME EST DEFINI
?
? POUR DEGRE :MON ; retourne le degre du monome :MON
> SI NOMBREP :MON [RT 0]
> RT DEGRE :MON
> FIN
DEGRE EST DEFINI
?
? POUR DEGRE :MON ; dans le cas d'un monome de degre > 0
> SI VIDE P :MON [RT 1]
> SI NOMBREP :MON [RT :MON]
> RT DEGRE SP :MON
> FIN
DEGRE EST DEFINI
?
? POUR COEF :MON
> SI VIDE P :MON [RT 1]
> SI MEMBREP :MON [+ -] [RT MOT :MON 1]
> SI NOMBREP :MON [RT :MON]
> RT COEF SD :MON ; on avance de droite a gauche.
> FIN
COEF EST DEFINI
?
? POUR COMPL :A :I ; completer :A par :I zeros a la fin.
> SI :I = 0 [RT :A]
> RT COMPL METD 0 :A :I-1
> FIN
COMPL EST DEFINI
?
? MONTRE FORM [2X3-4X+2]
[2 0 -4 2]

```


Voyons rapidement les fonctions de chaque procédure:

- Les procédures DEGRE et DEGREN retournent le degré d'un monome;
- DEGREN n'intervient que dans le cas où le degré n'est pas nul.
- La procédure COEF retourne évidemment le coefficient d'un monome.
- La procédure COMPL rajoute à la fin de la liste A autant de zéros que I l'indique.

Examinons la procédure FORME qui est plus élaborée. Le polynôme lu au clavier est d'abord reçu par LOGO qui va en séparer les monomes, les caractères + et - étant "speciaux", en ce sens que LOGO insère automatiquement un "blanc" après chacun d'eux.

Le polynôme P étant composé de monomes, la procédure FORME traite chaque monome l'un après l'autre jusqu'à épuisement. (ligne 1)

Dans les lignes 2, 3 et 4 on extrait le premier monome de P et on normalise P de sorte que dans tous les cas le premier monome occupe les deux premiers éléments de P.

La ligne 5 traite le cas du premier monome du polynôme P

La ligne 6 vérifie que le polynôme a bien été donné suivant les puissances décroissantes de X, sinon retour à l'expéditeur... (on pourrait trier à la main dans une version ultérieure).

La ligne 7 rajoute à P1 le nombre de zéros nécessaires suivant la différence des degrés de deux monomes consécutifs.

La ligne 8 met le coefficient du monome à la place convenable dans P1.

La ligne 9 mémorise le degré du dernier monome.

La ligne 10 part à la recherche du monome suivant.

OUF! que de travail pour transformer un brave polynôme tel $2X^3+3X-1$ en la liste [2 0 3 -1] ! Mais un oubli est si vite arrivé...

Continuons notre chemin en dérivant:

```
? POUR DERIVE :A
> RT DECODAGE DERIVEE FORM :A                ; coder, dériver, decoder !
> FIN
DERIVEE EST DEFINI
?
? POUR DERIVEE :P
> LOCAL "DEG
> DONNE "DEG (COMPTE :P) - 1                  ; COMPTE donne le nombre
> SI :DEG = 0 [RT []]                          ; d'éléments de :P
> RT METF :DEG * PREMIER :P DERIVEE SP :P
> FIN
DERIVEE EST DEFINI
?
? POUR DECODAGE :P
> RT ECRITURE :P []
> FIN
DECODAGE EST DEFINI
?
? POUR ECRITURE :A :B
> LOCAL "I                                     ; :I = 1 + degré(polynôme)
> SI VIDEP :A [SI VIDEP :B [RT 0] [RT :B]]      ; [] = le polynôme nul.
> DONNE "I COMPTE :A
> SI PREMIER :A=0 [RT ECRITURE SP :A :B ]
> DONNE "B METD MONEC PREMIER :A :I-1 :B
> RT ECRITURE SP :A :B
> FIN
ECRITURE EST DEFINI
?
? POUR MONEC :A :B
> SI :B = 0 [SI :A < 0 [RT :A] [RT MOT "+" :A]]
> SI :A > 0 [DONNE "A MOT "+" :A]
> SI :A = "/" + 1 [DONNE "A "+" ]              ; "/" + 1 fait que "+" est un opérateur
> SI :A = -1 [DONNE "A "-]                      ; unaire
> SI :B = 1 [RT MOT :A "X"]
> RT MOT (MOT :A "X) :B
> FIN
MONEC EST DEFINI
```

```
?
? EC DERIVE [-4X3+2X+5]
-12X2+2
```

Les procédures DERIVE et DERIVEE n'offrent pas de difficultés. Remarquons seulement que la procédure MONEC a pour premier paramètre le coefficient et pour deuxième paramètre le degré d'un monôme que l'on retourne en clair.

De même, la procédure ECRITURE reçoit une liste A représentant la suite associée à un polynôme et retourne la liste B qui est l'écriture habituelle du polynôme A.

Et la somme alors ?...

```
? POUR SOMMER :A :B
> RT DECODAGE SOM FORM :A FORM :B
> FIN
SOMMER EST DEFINI
?
? POUR SOM :A :B
> (LOCAL "CA "CB "S )
> DONNE "CA COMPTE :A
> DONNE "CB COMPTE :B
> SI :CA > :CB [DONNE "B COMPL2 :B :CA--:CB] [DONNE "A COMPL2 :A :CB--:CA]
> RT ADD :A :B []
> FIN
SOM EST DEFINI
?
? POUR COMPL2 :A :I ; completer :A par :I zeros en tete.
> SI :I = 0 [RT :A]
> RT COMPL2 METF 0 :A :I-1
> FIN
COMPL2 EST DEFINI
?
? POUR ADD :A :B :S
> SI VIDEP :A [RT :S]
> DONNE "S METD (PREMIER :A)+ PREMIER :B :S
> RT ADD SP :A SP :B :S
> FIN
ADD EST DEFINI
?
? EC SOMMER [2X3+4X2-2] [-3X4+X2-X+3]
? -3X4+2X3+5X2-X+1
```

La procédure COMPL2 met I zeros devant la liste A, tandis que la procédure SOM met d'abord à niveau les deux polynômes à additionner puis fait appel à ADD pour faire cette addition.

Enfin pour terminer occupons-nous du produit:

```
? POUR MULTIP :A :B
> RT DECODAGE PROD FORM :A FORM :B
> FIN
MULTIP EST DEFINI
?
? POUR PROD :A :B
> (LOCAL "R "LO )
> SI OU :A=0 :B=0 [RT 0]
> SI COMPTE :A < COMPTE :B [RT PROD :B :A]
> DONNE "LO -1 + (COMPTE :A) + COMPTE :B ;LO est le nombre d'elements
> DONNE "A COMPL :A :LO-COMPTE :A ;du polynome.On met le nombre
> DONNE "R COMPL [] :LO ;necessaire de zeros a A et R
> RT MULT METD PREMIER :A SP :A :B :R
> FIN
PROD EST DEFINI
```

```

? POUR MULT :A :B :R          ;R=A*B ,A et B ont le nombre de zeros desire
> SI VIDEP :A [RT :R]
> DONNE "A METF DERNIER :A SD :A      ;rotation circulaire de A
> SI PREMIER :B = 0 [RT MULT :A SP :B :R]
> DONNE "R ADD :R MULTI PREMIER :B :A [] [] ;on fait une addition
> RT MULT :A SP :B :R          ;partielle
> FIN
MULT EST DEFINI
?
? POUR MULTI :N :A :B          ;B=N*A ,N est un nombre,A et B sont des
> SI VIDEP :A [RT :B]          ;polynomes
> DONNE "B METD :N*PREMIER :A :B
> RT MULTI :N SP :A :B
> FIN
MULTI EST DEFINI
?
? EC MULTIP [4X3+2X+2] [-2X4-6X2-6X]
8X5-2X4+4X3-2X2-6X

```

La procedure PROD commence par prevoir par LD la longueur du polynome produit puis complete chaque polynome avec des zeros puis on passe la main a la procedure MULT qui, s'aidant de la procedure MULTI pour multiplier un polynome par un nombre, va realiser le produit des polynomes par decalages successifs, en commençant par les monomes de plus haut degre , comme nous le ferions avec nos bons vieux papier et crayon !
Nous voici enfin au bout de nos peines...

=====

II.2.5 Jeux recursifs

=====

1) Le probleme des N Dames

Prenons un echiquier de N cases par cote et munissons-nous de N dames. Il faut trouver toutes les configurations des N Dames sur l'echiquier de telle sorte qu'aucune d'elles ne puisse en "prendre" une autre. Les regles usuelles des Echecs sont utilisees.

Il apparait immediatement que le probleme a N dames n'a aucune solution pour $N < 4$. De plus, chaque ligne et chaque colonne contiendra exactement une Dame.

Le probleme n'est pas simple, dans le sens ou il n'y a pas de methode permettant de donner directement des solutions. Il nous faudra proceder par tatonnements, lequel devra neanmoins etre rigoureux de sorte a pouvoir affirmer a la fin que nous sommes bien en possession de toutes les solutions.

Sachant qu'une seule reine n'est possible par colonne, nous pouvons utiliser un pointeur sur les colonnes, qui sera incremente chaque fois qu'une position acceptable a ete trouvee sur une ligne de cette colonne. Ceci nous permettra de connaitre a chaque instant la colonne sur laquelle nous travaillons, ce qui sera utile pour effectuer les recherches de la prochaine ligne. Un pointeur analogue sur la ligne courante est utilise.

Nous allons partir d'un echiquier possedant une Dame en position (1,1) en haut a gauche. Les pointeurs de ligne et colonne ont donc la valeur 1 puis nous placerons une a une les N-1 Dames restantes en selectionnant leurs positions grace aux regles du jeu et a l'incrementation des pointeurs.

Après avoir incremente les pointeurs, nous determinerons la ligne, la colonne et les diagonales interdites par la case representee par les pointeurs. Si nos recherches montrent que la case choisie est acceptable (ne contient aucune Dame deja posee), alors nous pouvons y deposer une nouvelle Dame.

Si nos recherches montrent que la position n'est pas acceptable, il peut y avoir plusieurs cas:

- ou bien la colonne n'est pas finie, et alors nous incrementons le pointeur des lignes et poursuivons notre avance;
- ou bien nous sommes au bas de la colonne actuelle: cul-de-sac !!! Il faut alors remonter a la Dame precedemment posee, sur la colonne immediatement inferieure, incrementer les pointeurs a-partir de cette position, et reprendre les recherches.

Essayons de voir pourquoi cette position peut etre un cul-de-sac. Nous avons, a l'interieur d'une meme colonne, essaye toutes les positions possibles, et aucune ne s'est revelee efficace. L'echiquier actuel ne permet donc pas de placer de Dame sur cette colonne; il nous faut donc modifier cet echiquier. Mais ceci necessite une derniere precaution: si nous etions sur la colonne 1, alors il n'est plus possible de remonter a la Dame precedente: c'est donc la fin, nous avons trouve toutes les solutions possibles.

Enfin, au cours de nos recherches, nous serons en possession d'une solution si nous venons de placer la N-ieme Dame sur l'echiquier, et alors nous affichons l'echiquier.

Cette methode permet de minimiser le nombre de recherches pour aboutir aux solutions: des que nous trouvons une colonne ou il est impossible de placer une nouvelle Dame, nous remodelons l'échiquier, ce qui permet de ne pas etudier tous les échiquiers engendres par l'amorce d'échiquier que nous etudions. L'economie ainsi faite est enorme, le probleme etant insupportable meme par un ordinateur , si N est grand , et s'il fallait envisager tous les échiquiers possibles.

- * Ce style de resolution de problemes est nomme "programmation par
- * "back-track" i.e. retour-arriere . Elle permet de reduire consi-
- * derablement l'explosion combinatoire qui resulterait d'une ex-
- * ploration complete de l'arbre des positions.

Le programme:

Il se decompose en 4 procedures , la principale etant 8DAMES, qui se charge d'initialiser les variables, puis appelle BOUCLE qui va gerer le deroulement de la recherche.

```
POUR 8DAMES :TAILLE
(LOCAL "ECHIQUIER "N "M "HORIZ "VERT)
DONNE "VERT 0 DONNE "HORIZ 0
VIDETEXTE
DONNE "N 1 DONNE "ECHIQUIER []
DONNE "M 1
BOUCLE
FIN
```

```
POUR BOUCLE
SI CONFLIT :N :M :ECHIQUIER [ECHECM STOP]
DONNE "ECHIQUIER METD :M :ECHIQUIER
SI :N + 1 > :TAILLE [AFFICHE :ECHIQUIER ECHECN] [DONNE "N :N + 1 DONNE "M 1]
BOUCLE
FIN
```

```
POUR ECHECM
DONNE "M :M + 1
SI :M > :TAILLE [ECHECN]
BOUCLE
FIN
```

```
POUR ECHECN
TEST VIDE :ECHIQUIER
SIV [FIXCURSEUR PH 0 :VERT + :TAILLE EC "FINI RENVOIE "NIVEAUSUP]
DONNE "M (DERNIER :ECHIQUIER) + 1
DONNE "N COMPTE :ECHIQUIER
DONNE "ECHIQUIER SD :ECHIQUIER
SI :M > :TAILLE [ECHECN]
FIN
```

```
POUR AFFICHE :LIST
SI :HORIZ + :TAILLE > 39 [DONNE "VERT :VERT + :TAILLE + 1 DONNE "HORIZ 0]
SI :VERT + :TAILLE > 24 [DONNE "VERT 0 DONNE "HORIZ 0]
FIXCURSEUR PH :HORIZ :VERT
REPETE :TAILLE [REPETE :TAILLE [TAPE "-"] FIXCURSEUR PH :HORIZ (DERNIER
CURSEUR) + 1]
AFFICH1 :LIST
DONNE "HORIZ :HORIZ + :TAILLE + 1
FIN
```

```

POUR AFFICH1 :LIST
SI VIDE1 :LIST [STOP]
FIXCURSEUR PH (COMPTE :LIST) + :HORIZ - 1 (DERNIER :LIST) + :VERT - 1
TAPE "O AFFICH1 SD :LIST
FIN

POUR CONFLIT :N :M :DAMIER
SI VIDE1 :DAMIER [RT "FAUX]
RT OU (PEUR :N :M COMPTE :DAMIER DERNIER :DAMIER) (CONFLIT :N :M SD :DAMIER)
FIN

POUR PEUR :I :J :A :B
RT OU (:I = :A) OU (:J = :B) OU (:I - :J = :A - :B) (:I + :J = :A + :B)
FIN

```

```

--O--
O--
--O--
O--

```

```

--O--
--O--
O--
--O--

```

[2 4 1 3]

[3 1 4 2]

LE PROBLEME DES 4 DAMES

```

O----- O----- --O--- --O--- -O----- --O---
--O--- --O--- O----- O----- --O--- --O---
-O----- --O--- --O--- --O--- O----- O-----
---O--- -O----- -O----- --O--- --O--- --O---
--O--- --O--- ---O--- -O----- ---O--- -O-----

```

```

-O----- ---O--- --O--- --O---
--O--- -O----- -O----- --O---
--O--- --O--- --O--- -O-----
O----- O----- --O--- --O---
--O--- --O--- O----- O-----

```

LE PROBLEME DES 5 DAMES

2) Les Tours de Hanoi

Imaginons que nous avons trois tiges A, B, C devant nous. Sur la tige A se trouvent empilés N disques par tailles décroissantes. Le but du jeu est de déplacer tous les disques de la tige A sur la tige C par l'intermédiaire de la tige B, sous les conditions suivantes:

- on ne peut déplacer que le disque au sommet de chaque pile;
- on ne peut poser un disque que sur un disque plus grand.

Le problème est moins simple avec ces règles ! Il est relativement facile de s'amuser à déplacer un petit nombre de disques, mais la complication arrive vite, dès que N dépasse 4 ou 5. Nous allons développer une méthode générale (récursive) qui résolve le problème à N disques, ainsi que sa visualisation sur l'écran graphique.

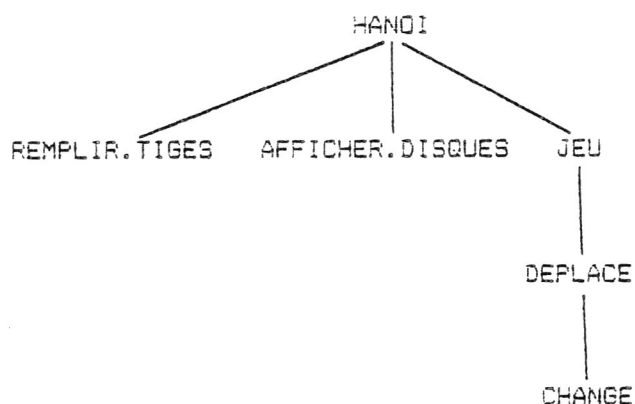
Si l'on observe la succession des coups à jouer, pour résoudre le problème à 6 disques, celle-ci peut paraître désordonnée. Regardons la situation générale avec N disques.

Il apparaît que pour déplacer N disques de A vers C, il suffit (!) de déplacer les N-1 premiers disques de A vers B, puis le N^{ème} vers C, et enfin de déplacer les N-1 disques de B vers C. Le problème à N disques est donc en principe résolu dès que l'on sait résoudre le problème analogue à N-1 disques. Les possibilités récursives de LOGO vont nous permettre de réaliser un programme basé sur ce principe, et chargé de déplacer N disques de la tige DEPART vers la tige ARRIVEE en utilisant la tige INTERMED comme intermédiaire:

- s'il n'y a qu'un seul disque, le déplacer directement;
- s'il y a N>1 disques:
 - 1) déplacer N-1 disques de DEPART vers INTERMED,
 - 2) déplacer le disque restant de DEPART vers ARRIVEE,
 - 3) déplacer les N-1 disques de INTERMED vers ARRIVEE.

Le programme comporte ici 3 procédures importantes qui sont appelées par HANOI. La première, REMPLIR.TIGES, permet de "remplir" les tiges avec les valeurs de départ (une seule est pleine, les autres sont vides). La procédure AFFICHER.DISQUES sert à préparer l'écran et à afficher les tiges dans leurs situations initiales.

Enfin, la procédure qui applique l'algorithme est nommée JEU. A l'intérieur de celle-ci est appelée DEPLACE qui gère les listes représentant les 3 tours, la représentation graphique étant, elle, assurée par CHANGE.



```

POUR HANOI :NOMBRE
  (LOCAL "UN "DEUX "TROIS)
  REMPLIR.TIGES :NOMBRE
  AFFICHER.DISQUES :NOMBRE
  JEU :NOMBRE "UN "TROIS "DEUX
    ; la procedure principale ne fait qu'appeler les autres. On va deplacer
    ; :NOMBRE disques de la tige 1 vers la tige 3 en passant par la tige 2.
    ; Ces parametres sont essentiels pour les appels recursifs de la proce-
    ; dure JEU.
  FIN

```

```

FIN

```

```

POUR REMPLIR.TIGES :N
  DONNE "UN [] ; puis la remplir...
  REPETE :N [DONNE "UN METF :N :UN DONNE "N :N - 1]
  DONNE "DEUX [] ; vides...
  DONNE "TROIS [] ; ...au depart.
  FIN

```

```

POUR AFFICHER.DISQUES :N
  VE CT ; initialise l'ecran graphique.
  LC FPOS [-70 -70] BC AV 80 ; tige 1
  LC FPOS [0 -70] BC AV 80 ; " 2
  LC FPOS [70 -70] BC AV 80 ; " 3.
  LC FPOS [-70 -70]
  REPETE :N [BC DR 90 RE 3 * :N AV 6 * :N RE 3 * :N LC GA 90 AV 10 DONNE "N :N
  - 1] ; dessin des N disques.
  FIN

```

```

POUR JEU :NB :DE :A :PAR
  SI :NB = 1 [DEPLACE :DE :A STOP]
  JEU :NB - 1 :DE :PAR :A
  JEU 1 :DE :A :PAR
  JEU :NB - 1 :PAR :A :DE
  FIN

```

```

POUR DEPLACE :DEP :ARR
  LOCAL "DISQUE
    ; on cherche quel est le disque a deplacer:
  DONNE "DISQUE PREMIER CHOSE :DEP
  DONNE :DEP SP CHOSE :DEP
    ; puis on supprime ce disque sur la tour de depart:
  DONNE :ARR METF :DISQUE CHOSE :ARR
    ; enfin, on deplace les disques a l'ecran:
  CHANGE :DISQUE :DEP :ARR
  FIN

```

```

POUR CHANGE :N :D :A
  (LOCAL "T "Q "D1 "A1)
    ; on prend les variables:
    T = hauteur de la tour de depart
    Q = hauteur de la tour d'arrivee
    D1 = le numero de la tour de depart
    A1 = le numero de la tour d'arrivee.
  DONNE "T COMPTE CHOSE :D
  DONNE "Q COMPTE CHOSE :A
  DONNE "D1 SI :D = "UN [1] [SI :D = "DEUX [2] [3]] ; * la conditionnelle dans
  DONNE "A1 SI :A = "UN [1] [SI :A = "DEUX [2] [3]] ; * une valeur calculee...
    ; on se positionne sur le disque D a deplacer:
  FPOS PH 70 * (:D1 - 1) - 70 :T * 10 - 70 FCAP 90

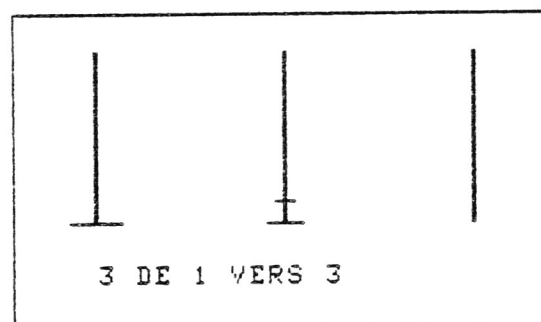
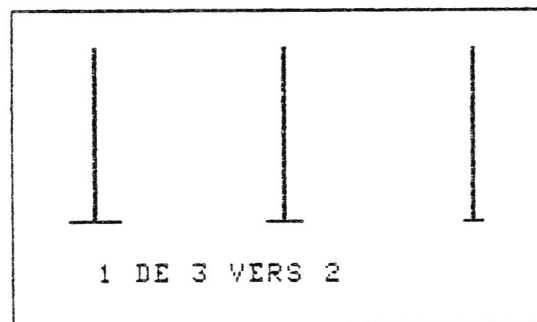
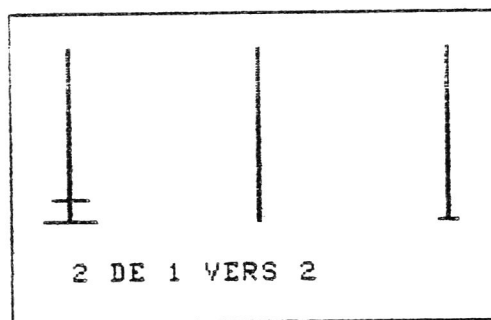
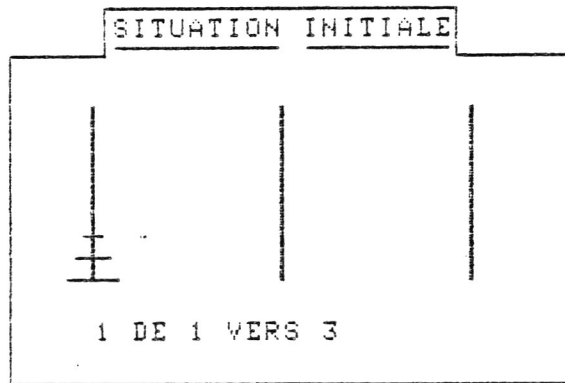
```

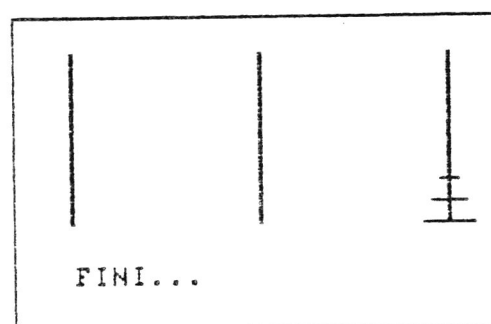
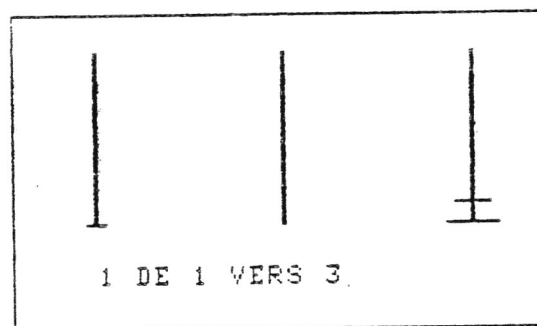
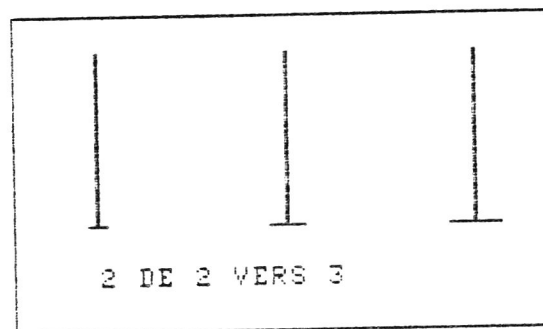
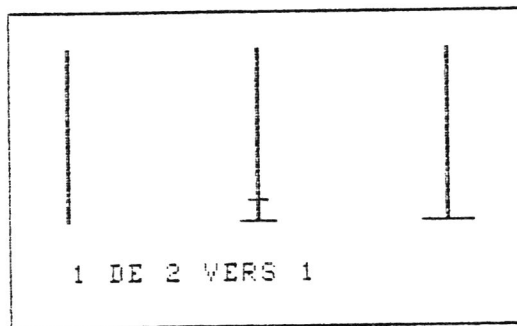


```

; on le gomme...
GC RE 3 * :N AV 6 * :N RE 3 * :N LC
; ...on redessine la partie effacee de la tige...
BC FCAP 0 RE 1 AV 2 RE 1 FCAP 90 LC
; ...on se positionne sur l'emplacement du disque d'arrivee...
FPOS FH 70 * (:A1 - 1) - 70 (:Q - 1) * 10 - 70
; ...et on le dessine.
BC RE 3 * :N AV 6 * :N RE 3 * :N LC
; Information pour le spectateur:
(ED :N "DE :D "VERS :A)
FIN

```





3) Le Baguenaudier

Nous disposons de N pions, disposés dans des cases devant nous. Le but du jeu est de tous les enlever, en respectant les règles suivantes:

- on peut à tout instant mettre ou enlever le pion de la 1ère case (celle qui est le plus à gauche).
- on peut mettre ou enlever un pion à une place telle qu'il y ait un pion dans la case immédiatement à sa gauche, et que ce soit le seul pion à sa gauche.

Procedons à quelques essais avec 4 pions:

```

initial :   X X X X
par b) :   X   X X
"  a) :       X X
           X   ; selon c
           X   X ; selon b
           X X X ; selon c

```

Nous avons réduit le jeu à 3 pions, ce qui est typique d'une démarche recursive:

- enlever le N-ième pion;
- résoudre le problème avec N-1 pions.

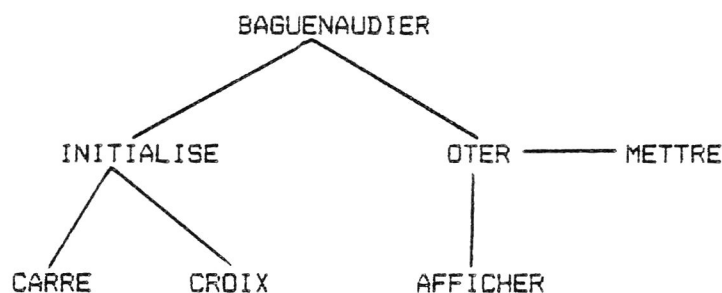
Le seul problème qui n'est pas résolu est: comment enlever le N-ième pion ?

Nous proposons d'enlever les N-2 premiers pions (faisable par hypothèse), puis, en vertu de la règle b), d'oter le dernier pion, et enfin de remettre les N-2 pions supprimés. Qui a dit bizarre ?...

Ce qui pose un nouveau problème : comment reconstituer un jeu de M pions ($M=N-2$) à gauche d'un pion donné ?

La encore, la technique est simple: reposons le dernier pion, puis appelons recursivement cette procédure sur les M-1 pions restants. Pour pouvoir reposer le M-ième pion, nous commencerons par ramener les M-1 pions qui le précèdent, puis nous en supprimerons les M-2 premiers, et il ne restera plus qu'à poser le pion M.

Quelle artillerie recursive !...



LE BAGUENAUDIER

 ETAT INITIAL

 PAR a)

 b)

 a)

 b)

 PAR b)

 b)

 a)

 b)

 a)

 b)

et voici le jeu ramené à l'ordre 4...

 PAR a)

 b)

 a)

 b)

 a)

...puis à l'ordre 3...

 PAR b)

 a)

 b)

...puis 2...

 a)

...1...

 b)

...et c'est fini !

 a)

LISTING DU PROGRAMME -----

La procedure centrale est BAGUENAUDIER qui appelle INITIALISE, afin de preparer l'ecran, puis elle demande a BAG d'oter les pions. Un jeu de va-et-vient s'installe alors entre BAG et DEBAG, cette derniere etant chargee de remettre des pions (procedures "corecursives").

Les Procedures AFFICHER et CROIX s'occupent de la gestion du mouvement des pions.

```
POUR BAGUENAUDIER :N
INITIALISE
OTER :N
FIN
```

```
POUR INITIALISE
VE FIXECHELLE 1
CT DR 90 BC AV 300
FPOS [-120 0]
REPETE :N [CARRE CROIX "PLEIN AV 10]
FIN
```

```
POUR CARRE
BC REPETE 4 [AV 10 GA 90] LC
FIN
```

```
POUR CROIX :CONTENU
FPOS PH XCOR + 5 YCOR + 5
SI EGALP :CONTENU "VIDE [GC] [BC]
REPETE 4 [AV 3 RE 3 DR 90]
LC FPOS PH XCOR + 5 0
FIN
```

```
POUR OTER :N
SI :N > 1 [OTER :N - 2 AFFICHER :N "VIDE METTRE :N - 2 OTER :N - 1] [SI
:N = 1 [AFFICHER :N "VIDE]]
FIN
```

```
POUR METTRE :N
SI :N > 1 [METTRE :N - 1 OTER :N - 2 AFFICHER :N "PLEIN METTRE :N - 2]
[SI :N = 1 [AFFICHER :N "PLEIN]]
FIN
```

```
POUR AFFICHER :N :CONTENU
FPOS PH 20 * :N - 140 0 CROIX :CONTENU
FIN
```

=====

II.2.6 QUELQUES TRIS DE LISTES

Pour illustrer encore un aspect de la recursivité, nous nous proposons de "trier" (par exemple en ordre croissant) une liste de nombres; typiquement:

```
? MONTRE TRI [3 -1 0 5 -2 10 8 -2]
[-2 -2 -1 0 3 5 10]
```

Il y a d'innombrables méthodes de tri; nous n'allons pas entrer dans les "analyses d'algorithmes" afin de comparer les performances respectives de chaque type de tri.

Présentons ici trois algorithmes de base:

- un tri "par insertion";
- un tri "par sélection";
- un tri "par fusion".

i) Un tri par insertion.

C'est un peu celui utilisé par les joueurs de cartes: on suppose un morceau de liste déjà trié, et on insère l'élément suivant dans la partie déjà triée (définition récursive).

L'algorithme suivant consiste à dire que:

- si la liste est vide, elle est toute triée;
- sinon, la liste triée s'obtient en insérant son premier élément dans le "saufpremier" trié auparavant:

```
? POUR TRI.INS :L
> SI VIDEP :L [RT []]
> RT INSERTION PREMIER :L TRI.INS SP :L
> FIN
TRI.INS EST DEFINI
? POUR INSERTION :EL :L ; pour insérer :EL dans :L
> SI VIDEP :L [RT LISTE :EL] ; en ordre croissant, la
> SI :EL < PREMIER :L [RT METF :EL :L] ; liste :L étant triée.
> RT METF PREMIER :L INSERTION :EL SP :L
> FIN
INSERTION EST DEFINI
```

ii) Un tri par sélection.

La aussi, le principe est récursif: on place le minimum en tête et on trie le reste (après avoir ôté le minimum en question):

```
? POUR TRI.SEL :L
> LOCAL "M
> DONNE "M MINIMUM :L
> RT METF :M TRI.SEL SUPPRESSION :M :L
> FIN
TRI.SEL EST DEFINI
?
? POUR MINIMUM :L
> SI VIDEP SP :L [RT PREMIER :L]
> RT MIN PREMIER :L MINIMUM SP :L
> FIN
MINIMUM EST DEFINI
?
? POUR MIN :X :Y
> SI :X < :Y [RT :X][RT :Y]
> FIN
MIN EST DEFINI
?
?
```

```

? POUR SUPPRESSION :EL :L
> SI :EL = PREMIER :L [RT SP :L]
> [RT PHRASE PREMIER :L SUPPRESSION :EL SP :L]
> FIN
SUPPRESSION EST DEFINI

```

iii) Un tri-fusion.

Encore un peu plus récursif, on coupe la liste en deux sous-listes de même taille, que l'on trie successivement, puis on mélange soigneusement les deux sous-listes ainsi triées. Le test d'arrêt de la récursion se fait sur les listes réduites à un élément. On supposera donc que la liste à trier n'est pas vide.

```

? POUR TRI.FUSION :L
> SI VIDE SP :L [RT :L]
> RT FUSIONNE TRI.FUSION MOITIE.GAUCHE :L TRI.FUSION MOITIE.DROITE :L
> FIN
TRI.FUSION EST DEFINI ; avec double appel récursif...
?
? POUR FUSIONNE :L1 :L2
> SI VIDE :L1 [RT :L2]
> SI VIDE :L2 [RT :L1]
> SI PREMIER :L1 < PREMIER :L2 [RT METF PREMIER :L1 FUSIONNE SP :L1 :L2]
> RT METF PREMIER :L2 FUSIONNE :L1 SP :L2
> FIN
FUSIONNE EST DEFINI
?
? POUR MOITIE.GAUCHE :L
? RT PREMIERS :L ENT (LONGUEUR :L) / 2 ; ENT = partie entière
? FIN
MOITIE.GAUCHE EST DEFINI
?
? POUR MOITIE.DROITE :L
> RT SFPREMIERS :L ENT (LONGUEUR :L) / 2
> FIN
MOITIE.DROITE EST DEFINI
?
? POUR LONGUEUR :L
> SI VIDE :L [RT 0]
> RT 1 + LONGUEUR SP :L
> FIN
LONGUEUR EST DEFINI
?
? POUR PREMIERS :L :N ; les :N premiers de :L
> SI :N = 0 [RT []]
> RT METF PREMIER :L PREMIERS SP :L :N - 1
> FIN
PREMIERS EST DEFINI
?
? POUR SFPREMIERS :L :N ; :L sauf ses :N premiers
> SI :N = 0 [RT :L]
> RT SFPREMIERS SP :L :N - 1
> FIN
SFPREMIERS EST DEFINI

```

II.2.7 Exercices et projets en LOGO.

=====

Quelques exercices pour maintenir la forme. Ils peuvent éventuellement reprendre certains thèmes déjà évoqués et/ou résolus, d'une autre manière.

Exercice 1 : Ecrire une fonction booléenne (i.e. retournant "VRAI" ou "FAUX")
----- "PLATE? :L" assurant qu'une liste :L est vide ou ne contient que des "mots":

```
? EC PLATE? [1 A 3 HELLO]
VRAI
? EC PLATE? [1 A [3] HELLO]
FAUX
```

Exercice 2 : Ecrire deux fonctions "SIGMA :L" et "MOY :L" retournant respectivement la somme et la moyenne des éléments d'une liste de nombres. La fonction MOY doit-elle obligatoirement utiliser SIGMA ?

Exercice 3 : Redéfinir une fonction "DER :L", qui doit retourner le dernier élément d'une liste ou la dernière lettre d'un mot:

```
? MONTRE DER [1 [2] [3 4]]
[3 4]
? EC DER 1234
4
? EC DER "SALUT
T
```

Exercice 4 : Ecrire une fonction "NIEME :N :L" retournant le N-ième élément de la liste :L ou la N-ième lettre du mot :L.

```
? MONTRE NIEME 3 [1 [2] [3 4] 5]
[3 4]
? EC NIEME 3 "SALUT
L
```

Exercice 5 : Ecrire une fonction "OCCUR :EL :L" retournant le nombre d'occurrences "au premier niveau" de l'élément :EL dans la liste :L, ou de la lettre :EL dans le mot :L. Autrement dit, on n'effectue aucune recherche à l'intérieur de chaque élément de :L.

```
? EC OCCUR 1 [2 1 [1 -1] 3 1 0]
2
```

Même problème pour rechercher le nombre d'occurrences à tous les niveaux:

```
? EC OCCUR# 1 [2 1 [-1 1 [1 -1]] 3 1 0]
4
```

Exercice 6 : Ecrire une fonction "PREMIERS :L" retournant la liste de tous les premiers éléments des listes éléments de :L.

```
? MONTRE PREMIERS [[1 2] 3 [HELLO LOGO] CA VA]
[1 HELLO]
```

Même problème avec DERNIERS.

Exercice 7 : Redéfinir la fonction booléenne "MEMBRE :EL :L" qui retourne
 ----- "VRAI si et seulement si :EL est élément (1er niveau) de :L.
 Même problème avec "MEMBRE# :EL :L" qui teste si :EL apparaît
 dans :L à un niveau quelconque.

Exercice 8 : Écrire une fonction "INSERTION :EL1 :EL2 :L" chargée de retour-
 ----- ner la liste :L dans laquelle l'élément :EL1 a été inséré après
 la première occurrence de l'élément :EL2.

Exercice 9 : Écrire une fonction "SUPPRESSION :EL :L" retournant la liste :L
 ----- privée de toutes les occurrences (au premier niveau) de l'élé-
 ment :EL. Même chose pour SUPPRESSION# à tous les niveaux.

Exercice 10: Définir une fonction "REUNION :L1 :L2" retournant la liste des
 ----- éléments (1er niveau) apparaissant dans :L1 ou dans :L2.
 Même problème avec "INTERSECTION :L1 :L2" retournant la liste
 des éléments apparaissant à la fois dans :L1 et dans :L2.
 Écrire une seule fonction pour chaque, à la différence de ce
 qui a été fait en II.2.1; quelle est la version la plus perfor-
 mante : en temps ? en mémoire ?

Exercice 11: Écrire une fonction "BOUTCOMMUN :L1 :L2" retournant la plus
 ----- longue liste commune aux extrémités droites des listes :L1 et
 :L2:

```
? MONTRE BOUTCOMMUN [A B 3 C 5 [G H]] [A B -1 2 C 5 [G H]]
[C 5 [G H]]
```

Écrire de même "RADICAL :M1 :M2" retournant le plus long mot
 commun aux extrémités gauches de deux mots :M1 et :M2:

```
? EC RADICAL "MONTRONS "MONTREZ
MONTR
```

Cette fonction peut être le point de départ d'un programme de
 conjugaison.

Exercice 12: Écrire une fonction "SUBST :L1 :L2 :L" retournant la liste :L
 ----- dans laquelle chaque occurrence de :L2 au premier niveau a été
 remplacée par :L1. On ne substitue donc pas à l'intérieur de
 chaque élément de :L. À noter que :L1 et :L2 peuvent être des
 listes:

```
? MONTRE SUBST "& "ET [ET P [ET Q R]]
[& P [ET Q R]]
```

Même chose que ci-dessus, en substituant à tous les niveaux de
 la liste:

```
? MONTRE SUBST# "& "ET [ET P [ET Q R]]
[& P [& Q R]]
```

Exercice 13: Écrire une fonction "APPLICATION :FCT :L" retournant la liste de
 ----- tous les résultats d'application de la fonction :FCT à chaque
 élément de la liste :L:

```
? MONTRE APPLICATION "CARRE [1 2 3] ; CARRE 2 = 4
[1 4 9]
```

Exercice 14: Ecrire une fonction "SELECT :EL :L", dans laquelle :L est une
 ----- liste [val11 val12 val21 val22 ...] , et :EL un objet (mot ou
 liste), la fonction SELECT retournant valj2, si j est le plus
 petit indice i tel que :EL=vali1. Plus précisément, si valj2 est
 un objet constant, cet objet est retourné, et si valj2 est de la
 forme :X, c'est la valeur effective de X qui est retournée, et
 non pas le mot ":X.

```
? DONNE "X 3 DONNE "Y "B DONNE "Z [1 2]
? EC SELECT :Y [A 5 B 4 [1 2] :X 7 -1]
4
? EC SELECT :Z [A 5 B 4 [1 2] :X 7 -1]
3
```

Exercice 15: Ecrire une procédure (pas une fonction) "RPLACPREM :L :L1" qui
 ----- remplace le premier élément de la liste :L par :L1 Cette procé-
 dure n'a donc rien à "retourner" et le prompt ? doit apparaître
 aussitôt après son exécution. Ecrire de même une RPLACSP devant
 remplacer le "saufpremier".

Cet exercice illustre bien la manière de faire un "effet de
 bord" sur l'un des paramètres d'une procédure LOGO. Ceci semble
 contredire à premier abord le fait que malgré toute modification
 d'un paramètre d'une procédure pendant l'exécution de celle-ci,
 le paramètre en question est retrouvé intact au sortir de la
 procédure:

```
? POUR ESSAI :N
> DONNE "N :N+1
> EC :N
> FIN
ESSAI EST DEFINI
? DONNE "N 1
? ESSAI :N
2          ; le changement...
? EC :N
1          ; ...dans la continuité.
```

Exercice 16: Generaliser l'exercice 5 en écrivant une procédure (encore...)
 ----- "RPLACN :N :L :L1" remplaçant le N-ième élément de :L par :L1.
 Ceci peut permettre aux gens pressés (ou conservateurs) de simu-
 ler immédiatement en LOGO des algorithmes écrits en fait pour
 des tableaux (affectations du style T[I] := X).

Exercice 17: Ecrire une fonction "EXPLOSE :M" prenant pour paramètre un mot
 ----- :M et retournant la liste des lettres de ce mot; écrire de même
 une fonction "IMPLOSE :L", réalisant l'opération inverse de la
 précédente:

```
? MONTRE EXPLOSE "HELLO
[H E L L O]
? EC IMPLOSE [L I S T E]
LISTE
```

L'extension des fonctions PREMIER et SAUFPREMIER aux mots est
 équivalente à la donnée des fonctions IMPLOSE et EXPLOSE . Le
 choix des implémenteurs de LOGO ne s'est pas porté sur ces fonc-
 tions d'explosion d'identificateurs, au prix d'une perte certai-
 ne de place en mémoire, si précieuse dans ce langage.

Exercice 18: Ecrire une fonction "INVERSE :L" qui retourne la liste :L dans
----- l'ordre inverse:

```
? MONTRE INVERSE [1 2 3 4]
[4 3 2 1]
```

Exercice 19: Ecrire une fonction "APLATIR :L" qui retourne la liste des mots
----- composant :L:

```
? MONTRE APLATIR [A [B [A D] E] F [G]]
[A B A D E F G]
```

Exercice 20: Ecrire une fonction "PERMUT :L" écrivant toutes les permutations
----- d'une liste d'éléments distincts:

```
? PERMUT [A B C]
ABC BAC CAB ACB CBA BCA
```

Tachez de l'écrire en une seule fonction...

Projet 1 : Pour les EA0logistes: implementer un systeme de calcul analyti-
===== que plan. Coordonnees, produit scalaire, compositions d'isome-
tries, etc...

Projet 2 : Amplifiez l'exercice 11 pour realiser un programme complet de
===== conjugaison (tous temps et personnes) . Attention aux verbes
irreguliers...et a la place en memoire.

Projet 3 : Implementer une Tortue en 3 dimensions, ou lire l'article sui-
===== vant, qui l'a deja realise (en Apple-LOGO anglais):

```
"LOGO: une Tortue en trois dimensions"
par Y. Orlarey
Micro-Systemes No 29, Mars 1983
```

Projet 4 : Ecrire un programme qui reconnaisse des phrases simples du type
===== [== est un(e) ==]
[== est-il(elle) un(e) == ?]

Le programme donnera une reponse (oui, non, vu, ne-sais-pas),
sur la base des phrases (connaissances) deja entrees:

```
> NICOLAS EST UN GENERAL
= VU
> UN GENERAL EST UN HOMME
= VU
> NICOLAS EST-IL UN HOMME ?
= OUI
> NICOLAS EST-IL REEL ?
= NE-SAIS-PAS
```

Ce programme peut donner des idees sur une possible gestion
"intelligente" d'une petite base de donnees.

Vous pouvez eventuellement utiliser la fonction FILTRE suivante
pour comparer la phrase entree avec celles qui sont disponibles
en memoire, mais ce n'est pas obligatoire.

Projet 5 : Ecrire une fonction "FILTRE :L1 :L2" chargée de dire si la liste :L1 (source) ressemble à la liste :L2 (but) au sens suivant:

```
? EC FILTRE [A B C] [A B C]
VRAI
? EC FILTRE [A [? X] C] [A B C]
VRAI      ; et de plus FILTRE aura fait un effet de bord sur X en
?          ; lui donnant la valeur "B. La seconde liste est donc
?          ; toujours constante, et la fonction FILTRE essaie de
?          ; faire coïncider la 1ère avec elle, au besoin en
?          ; cherchant à quelle condition. Encore un exemple:
? SI FILTRE [HELLO [? X] MACHIN] [HELLO DOCTEUR MACHIN] [EC :X]
DOCTEUR
? EC FILTRE [1 [? X] 2] [1 DOCTEUR 3]
FAUX
```

Vous pouvez pousser plus loin en autorisant [* X] pouvant remplacer non seulement un élément, mais toute une portion de liste comme dans l'exemple suivant:

```
? SI FILTRE [[* X] EN [* Y]] [NICOLAS B EN 1ERE SI] [(EC :X :Y)]
NICOLAS B 1ERE S
```

Cette fonction permettrait par exemple un dialogue souple entre l'utilisateur et la machine. Un programme d'EAO est la dernière chose au monde devant obéir à une syntaxe rigide. Peu d'efforts ont été faits en ce domaine (sur micros) pour se rapprocher du langage naturel.

Dans l'ordre de cette idée, et sur la technique plus générale de "filtrage" et de son utilisation dans l'étude de l'apprentissage de la programmation, consulter l'article:

"Quelques problèmes concernant l'adéquation des langages applicatifs pour la représentation de processus cognitifs"
par H. Wertz, in:
Actes de la 10-ième Ecole de Printemps
(Interpretation et Compilation)
Publications du LITP, Avril 1983
(2, place Jussieu, PARIS)

=====

CONFIDENTIAL

II.3 - Guidage récursif de la Tortue -

II.3.1 Polygones.

Montrez à un élève (débutant!) comment tracer un carré:

```
? POUR CARRE :C
> REPETE 4 [AV :C DR 90]
> FIN
CARRE EST DEFINI ; dont acte...
```

Pas de problème. Passez-lui le clavier, et proposez-lui de faire tracer par la Tortue un triangle équilatéral de côté :C. Il y a de fortes chances pour qu'il écrive:

```
? POUR EQUIL :C
> REPETE 3 [AV :C DR 60]
> FIN
EQUIL EST DEFINI
```

Dans quel monde sommes-nous?... Les angles d'un triangle équilatéral ne seraient-ils plus de 60 degrés? La confusion provient du fait que cette brave Tortue, pour tracer une figure, a besoin de connaître ses angles extérieurs, et non intérieurs, comme notre éducation mathématique, guère habituée à manipuler des "repères mobiles", nous a conditionné à le faire. L'erreur est vite rectifiée:

```
? POUR EQUI :C
> REPETE 3 [AV :C DR 120]
> FIN
EQUI EST DEFINI ; et fonctionne bien...
```

Les structures identiques des procédures CARRE et EQUI invitent à une généralisation immédiate: on remplace 3 ou 4 par un entier :N que l'on met en paramètre, et on en fait autant avec l'angle 90 ou 120 qui devient :A ce qui produit:

```
? POUR POLY :N :C :A
> REPETE :N [AV :C DR :A]
> FIN
POLY EST DEFINI
```

Et donne bien ce que l'on attend, mais le polygone obtenu est convexe ou croise suivant...

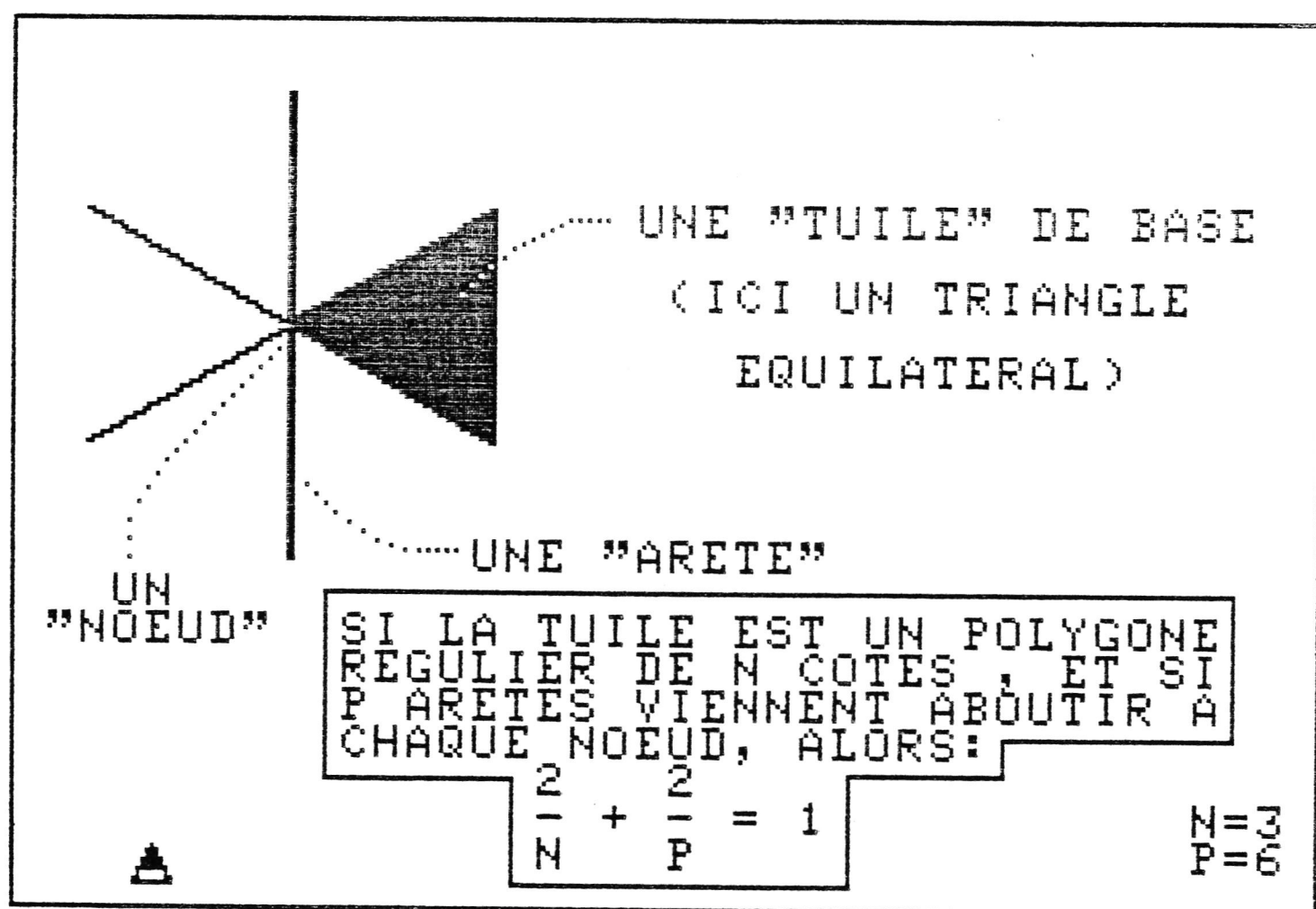
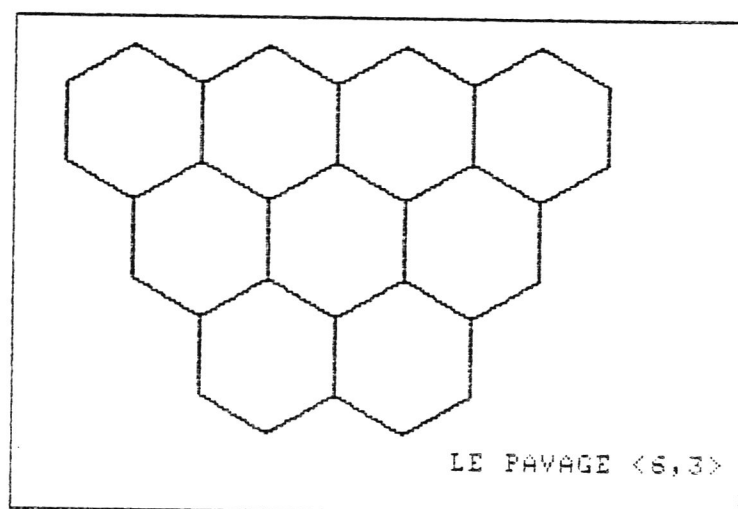
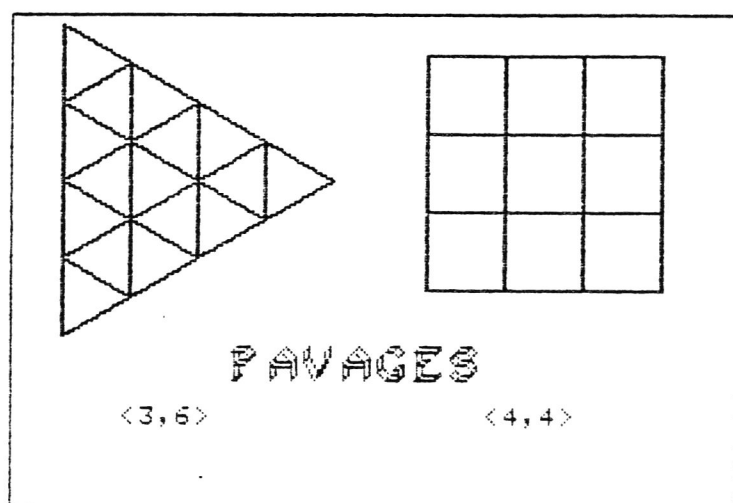
Pour l'élève: 1) suivant quoi ?...

- 2) la courbe obtenue se referme-t-elle toujours à son point de départ ? Envisager la valeur modulo 360 du produit :N * :A (ce dernier est l'ANGLE TOTAL dont a tourne la Tortue lors de son trace).

Une démonstration rigoureuse de l'équivalence:

polygone ferme $\Leftrightarrow$:N * :A = 0 [360]

peut être basée sur le fait que les sommets du polygone sont cocycliques et qu'à chaque étape la longueur imposée :C n'autorise que deux cordes possibles; donc dès que :N * :A devient multiple de 360, la Tortue est sur le cercle avec son cap initial, ce qui ne donne que deux points possibles sur le cercle: ou bien le point initial, ce qui démontre le résultat, ou bien son symétrique par rapport au diamètre dirige suivant le cap initial, auquel cas un second parcours - symétrique du premier - ramènerait bien la Tortue au point de départ.



Nous allons maintenant envisager une définition recursive de POLY basee sur la constatation que pour faire un polygone de parametres :C et :A, il suffit d'avancer de :C, de tourner a droite de :A, puis de recommencer. Notez que l'information du nombre :N de cotes est redondante, au moins tant que l'on ne tient pas compte d'un petit detail:

```
? POUR POLYREC :C :A
> AV :C DR :A
> POLYREC :C :A
> FIN
POLYREC EST DEFINI
```

Vous reconnaissez ici l'un des cas dont nous avons parle plus haut, de "recursion terminale"; le programme s'ecrirait, en Pascal iteratif:

```
Procedure POLYREC(C,A:integer);
Begin
  repeat
    move(C);
    turn(A)
  until false
End;
```

La version Pascal montre bien le "petit detail" qui risque de faire perdre patience: la procedure "boucle", la tortue, infatigable, parcourant son bonhomme de chemin meme si la courbe se referme!

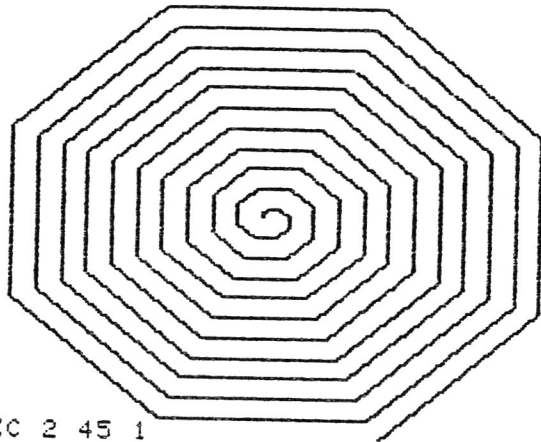
Tachons d'utiliser les remarques precedentes sur l'angle total pour forcer la procedure a stopper des que la Tortue se retrouve dans son etat initial (etat = (point , cap) au moins).

```
? POUR POLY :C :A
> POLYREC1 0
> FIN
POLYREC EST DEFINI
? POUR POLYREC1 :T ; :T = angle total
> AV :C DR :A
> SI RESTE :T+:A 360 = 0 [STOP]
> POLYREC1 :T+:A
> FIN
POLYREC1 EST DEFINI
```

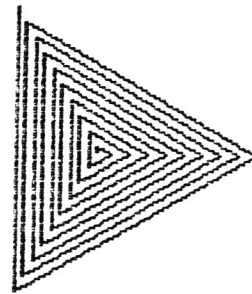
Remarquez que les parametres :C et :A, qui sont locaux a la procedure POLYREC, sont globaux pour la procedure POLYREC1; ceci n'est pas grave si POLYREC1 ne peut etre appelee que par POLYREC. Nous aurions pu, pour plus de precautions, les inclure aussi dans la liste des parametres.

Une investigation possible avec les eleves concernant les polygones reguliers consiste a montrer la finitude du nombre possible de pavages plans avec de tels polygones:

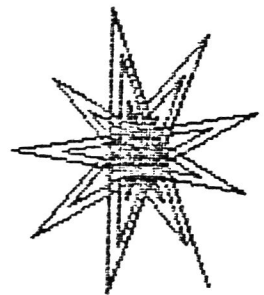
- Que vaut un angle interieur d'un polygone P_n regulier a n cotes ? (Le meme style de demonstration, utilisant l'angle total dont a tourne la Tortue pour le tracer, permet d'obtenir la somme totale des angles interieurs de P_n).
- Un pavage est de type $\langle n,p \rangle$ si les tuiles de base -toutes egales- sont des P_n et si a chaque "noeud" du pavage aboutissent p aretes. Montrer que la possibilite de paver equivaut a la relation suivante liant n et p : $2/n + 2/p = 1$.
- En deduire que les seuls pavages cherches sont les $\langle 3,6 \rangle, \langle 4,4 \rangle$, et $\langle 6,3 \rangle$; chacun a son "dual".
- En profiter pour demander a la Tortue de realiser un morceau de pavage $\langle n,p \rangle$ afin de visualiser l'impossibilite dans certains cas.



SPIREC 2 45 1

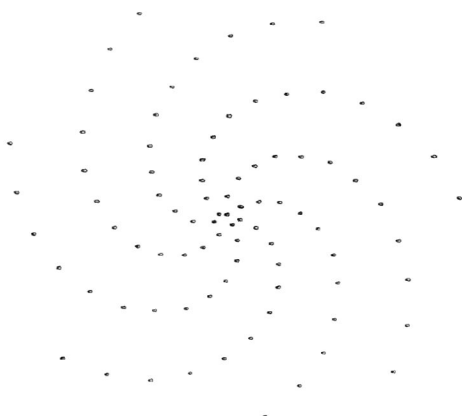


SPIREC 2 120 4



SPIREC 2 160 4

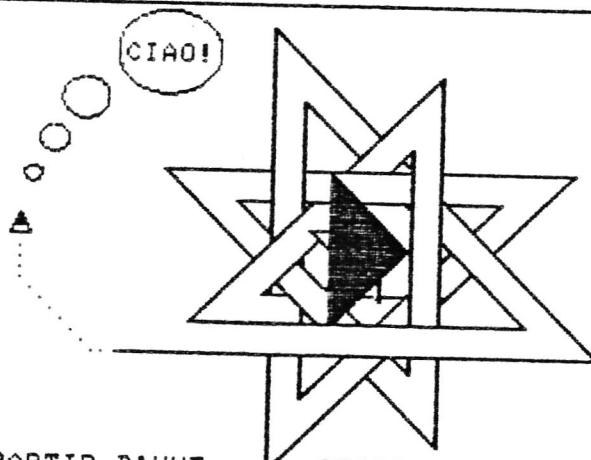
004-02-1400



SPIRECDUAL 2 156 2



SPIANG 10 10 1



(A PARTIR D'UNE.....SPIREC 3 135 7)

II.3.2 Spirales.

On peut visualiser une "figure spirale", au sens general du terme, comme une generalisation de la "figure polygonale"; reprenons en effet la procedure POLYREC, mais avec un appel recursif modifie de maniere a changer, a chaque "sommet" la "taille" (i.e. le(s) parametre(s)) de la figure:

```
? POUR SPIREC :C :A
> AV :C DR :A
> SPIREC :C+1 :A          ; recommencer en augmentant le cote
> FIN
SPIREC EST DEFINI
```

Les dessins reproduits montrent deja une belle variete de figures spirales, qui peuvent etre l'occasion de discuter avec les eleves du choix de tel ou tel angle pour obtenir tel ou tel type de figure.

Sur cette voie, on peut aussi etre tente de parametrier l'increment I (problemes de resolution sur l'ecran):

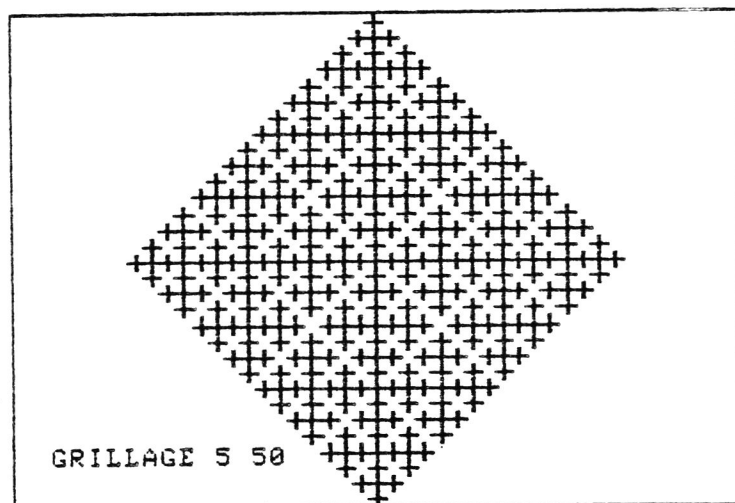
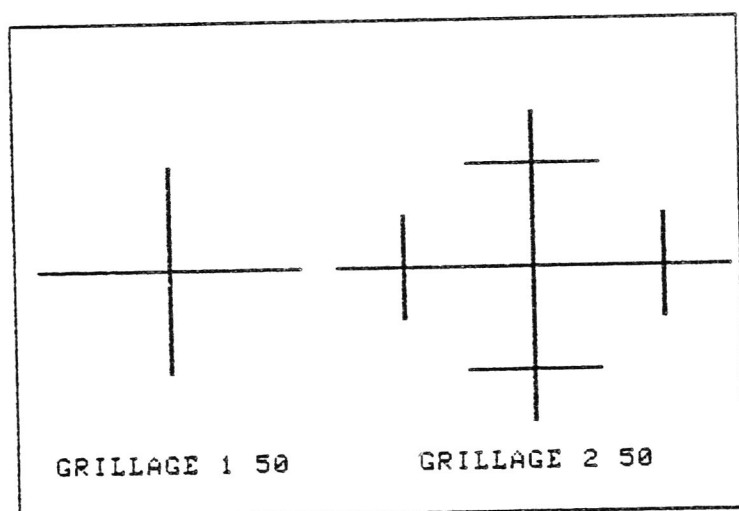
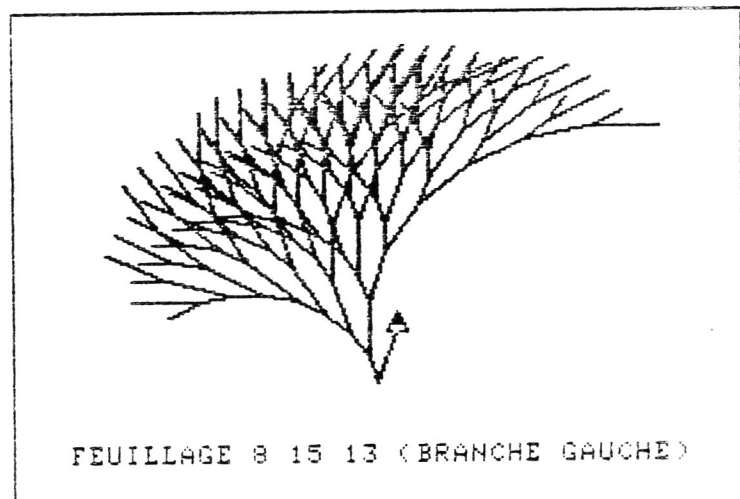
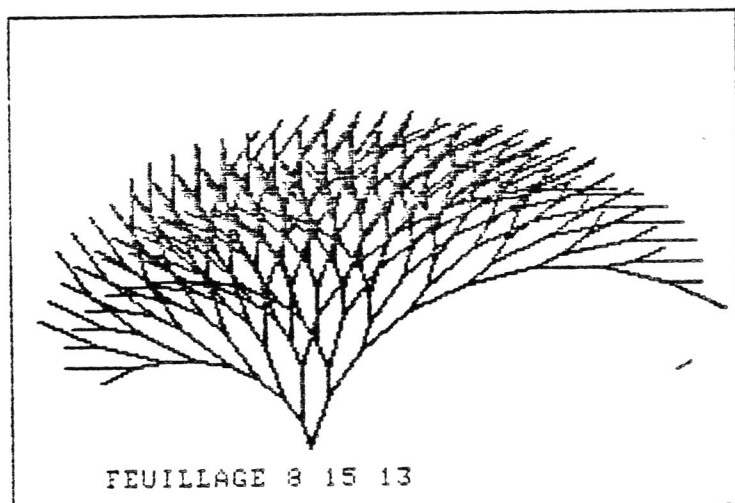
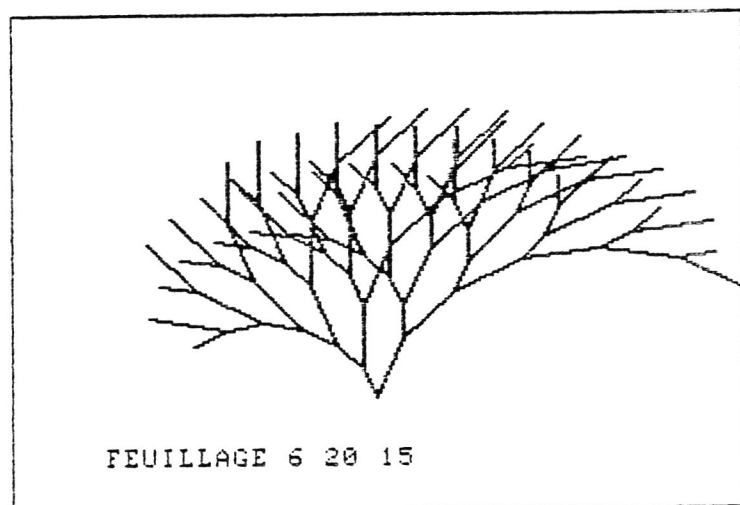
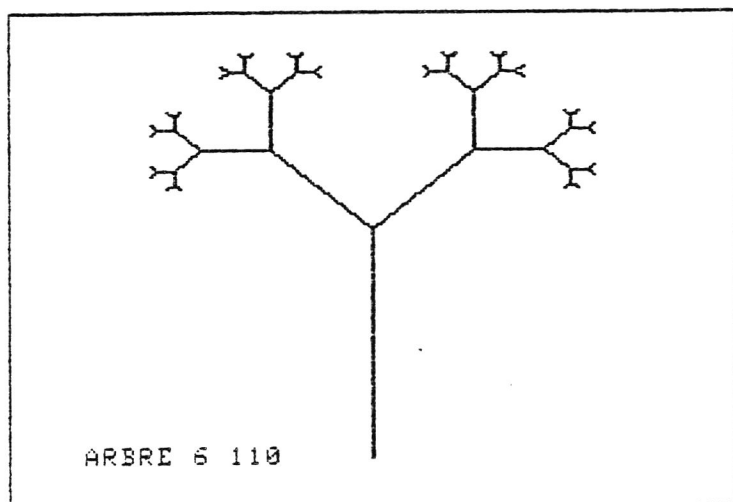
```
? POUR SPIREC1 :C :A :I
> AV :C DR :A
> SPIREC1 :C+:I :A :I
> FIN
SPIREC1 EST DEFINI
```

Comparez en effet SPIREC 3 120 et SPIREC1 3 120 5 qui sont topologiquement identiques, ce que n'indique pas l'ecran...Remarquez que ceci depend encore des machines, et meme de la couleur que vous utilisez pour certaines d'entre elles (c'est le cas de l'APPLE II, pour lequel chaque point de l'ecran graphique ne peut etre de n'importe quelle couleur; vous pouvez voir ci-contre la difference entre deux traces "identiques" avec des couleurs distinctes, blanc et orange). Un moniteur video noir et blanc peut permettre, avec ces couleurs, d'effectuer des "ombrages".

Il est interessant aussi d'observer en quelque sorte la figure "duale" obtenue en ne tracant que les sommets:

```
? POUR SPIRECDUAL :C :A :I
> LEVECRAYON
> AV :C DR :A
> BAISSSECRAYON
> POSEPOINT
> SPIRECDUAL :C+:I :A :I
> FIN
SPIRECDUAL EST DEFINI
? POUR POSEPOINT          ; uniquement pour la lisibilite
> POINT POS
> FIN
POSEPOINT EST DEFINI
```

La procedure POINT :COORD trace un seul point (de la couleur courante du crayon) sur l'ecran; :COORD est une liste dont les deux elements sont les coordonnees du point a tracer. Quant a POS, c'est precisement une fonction sans argument retournant la liste des deux coordonnees de la Tortue au moment de l'appel (depend peut-etre de l'implementation; c'est pourquoi nous avons introduit POSEPOINT).



Une autre possibilité est de faire varier l'angle au lieu du côté, pour modifier cette fois la "courbure" de la figure spirale:

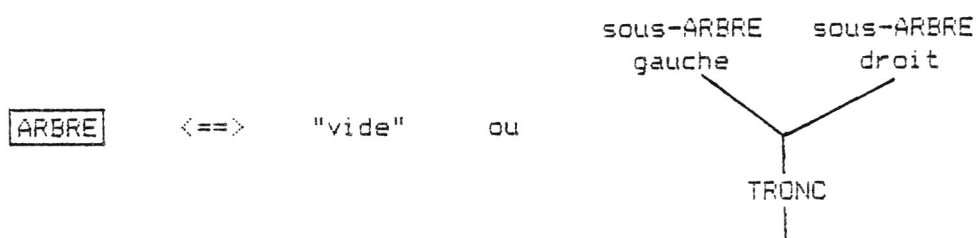
```
? POUR SPIANG :C :A :I
> AV :C DR :A
> SPIANG :C :A + :I :I
> FIN
SPIANG EST DEFINI
```

Pour l'élève: tacher d'expliquer les phénomènes de changement de courbure

 notamment au(x) "point(s) d'inflexion".

II.3.3 Arborescences.

Nous proposons la définition recursive suivante de la figure "ARBRE":

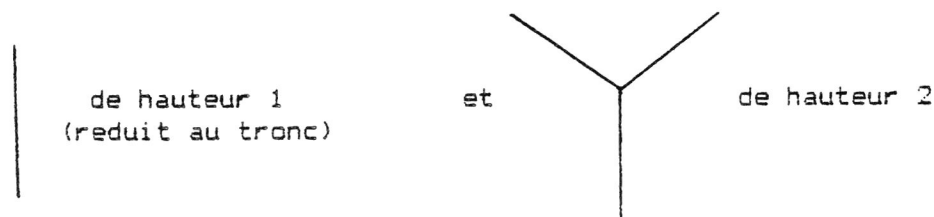


Ceci ne definit en fait que les "arbres binaires", i.e. tels qu'à chaque noeud, il ne sorte que deux nouvelles branches, ou aucune. Libre à vous d'expérimenter avec des arbres ternaires ou même n-aires... (un arbre unaire n'est autre qu'une ligne brisée).

La "hauteur" d'un tel arbre est elle aussi définie recursivement par:

- H("vide") = 0
- H(arbre) = 1 + Max(H(s.a. gauche), H(s.a. droit)).

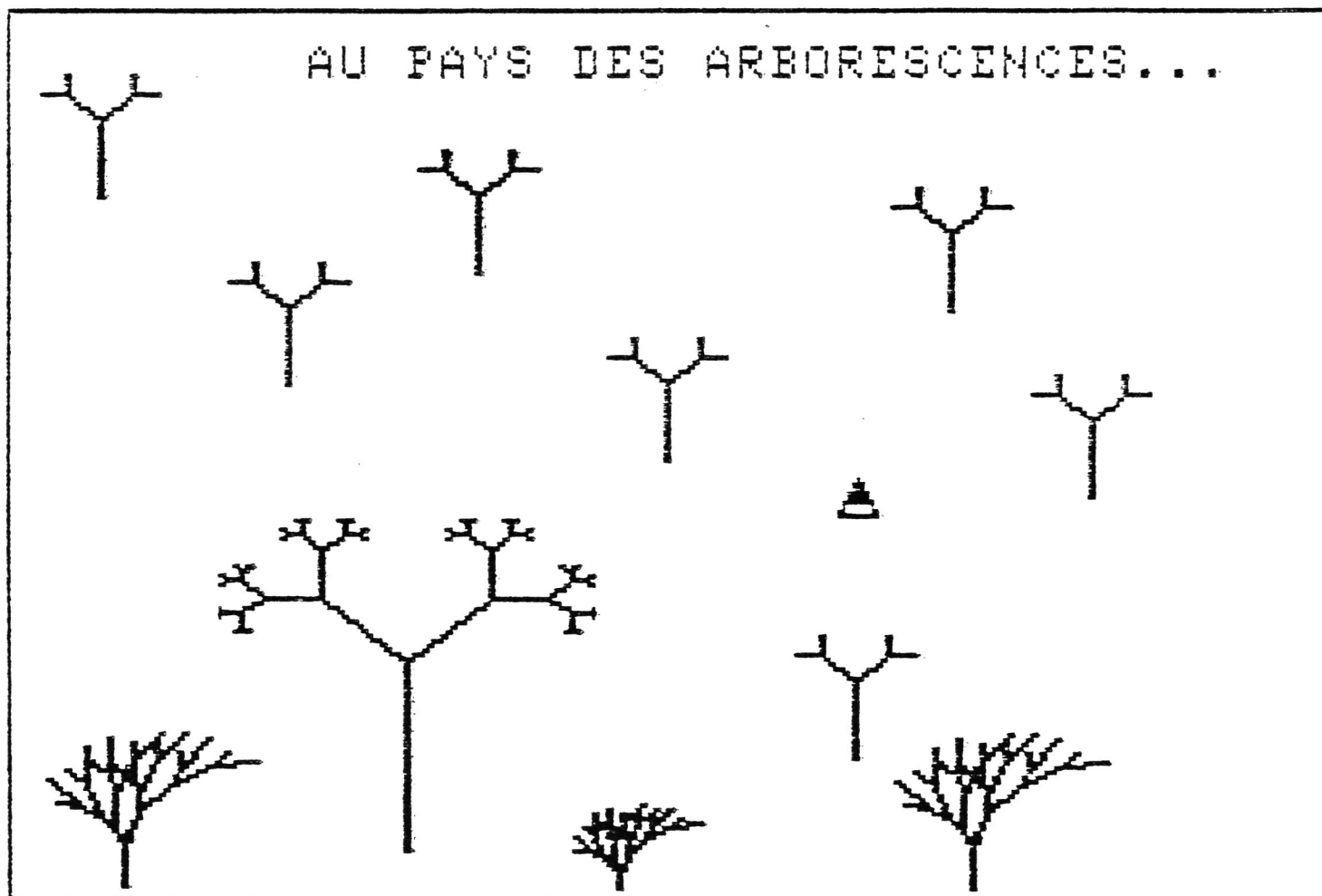
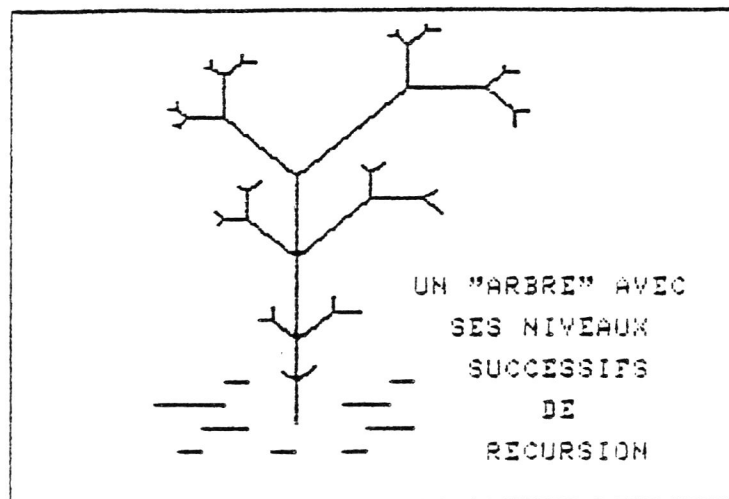
Par exemple:



Fidèle à la morale ambiante, la procédure ARBRE chargée du trace d'un tel objet défini recursivement, sera elle-même recursive. On exigera de plus que la Tortue retrouve son état initial à la fin du trace:

```
? POUR ARBRE :N :C ; :N est la hauteur de l'arbre.
> SI :N=0 [STOP] ; le vide au niveau 0.
> AV :C
> GA 45 ARBRE :N-1 :C/2 ; les...
> DR 90 ARBRE :N-1 :C/2 ; ...branches.
(*) > GA 45 RE :C ; pour l'invariance d'état.
> FIN
ARBRE EST DEFINI
```

La ligne (*) - essentielle - est souvent oubliée par les élèves... Elle permet aussi de prouver par récurrence sur :N l'invariance de l'état de la Tortue.



Bien entendu, la première généralisation va consister à passer l'angle de branchement en paramètre:

```
? POUR ARBRE.ANG :N :C :A
> SI :N=0 [STOP]
> AV :C
> GA :A ARBRE.ANG :N-1 :C/2 :A ; ou en modifiant :A...
> DR 2*:A ARBRE.ANG :N-1 :C/2 :A
> GA :A RE :C
> FIN
ARBRE.ANG EST DEFINI
```

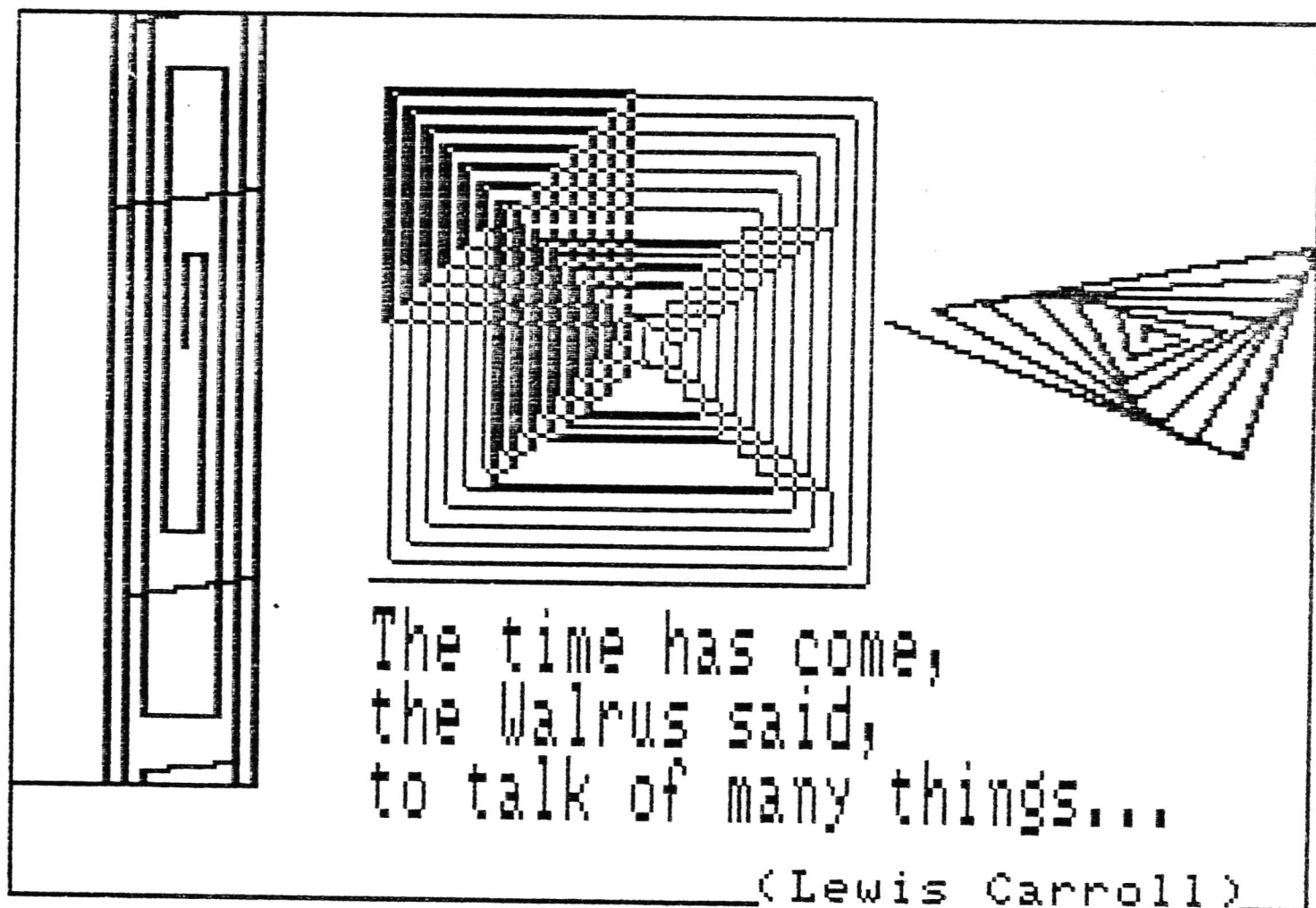
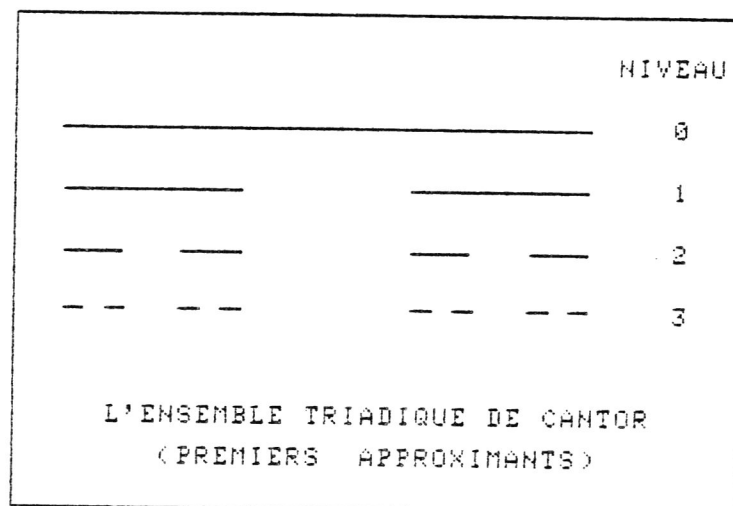
Pour l'élève: Quelle est la longueur totale du chemin parcouru par la
 ----- Tortue lors de ARBRE.ANG :N :C :A ? Depend-elle de :A ?
 Est-ce la même que celle de la courbe tracée ? Quelle est la
 longueur du "chemin limite", etc...

On peut essayer aussi d'éliminer la trop parfaite symétrie par-rapport au
 tronc principal, en traçant des sous-arbres gauche et droit de tailles diffé-
 rentes:

```
? POUR FEUILLAGE :N :C :A
> SI :N=0 [STOP]
> GA :A
> BRANCHE.GAUCHE :N-1 :C :A
> DR 2 * :A
> BRANCHE.DROITE :N-1 :C :A
> GA :A
> FIN
FEUILLAGE EST DEFINI
?
? POUR BRANCHE.GAUCHE :N :C :A
> AV :C
> FEUILLAGE :N :C :A
> RE :C
> FIN
BRANCHE.GAUCHE EST DEFINI
?
? POUR BRANCHE.DROITE :N :C :A
> AV 2 * :C ; par exemple...
> FEUILLAGE :N :C :A
> RE 2 * :C
> FIN
BRANCHE.DROITE EST DEFINI
```

Enfin, un exemple de courbe à plusieurs branchements est fournie par:

```
? POUR GRILLAGE :N :C
> SI :N=0 [STOP]
> AV :C
> GRILLAGE :N-1 :C/2
> RE 2*:C DR 180
> GRILLAGE :N-1 :C/2
> RE :C DR 90 AV :C
> GRILLAGE :N-1 :C/2
> RE 2*:C DR 180
> GRILLAGE :N-1 :C/2
> RE :C GA 90
> FIN
GRILLAGE EST DEFINI
```



Conseil: suivez pas a pas les instructions de la procedure precedente: elle montre bien le mecanisme de l'écriture recursive. Pour faire un grillage de niveau :N, on se place successivement sur les 4 points cardinaux, et la on fait chaque fois un grillage de niveau :N-1 (hypothese de recurrence graphique...), la recurrence butant sur le niveau 0.

La courbe limite remplit en fait tous les points d'un carre. Nous verrons au paragraphe suivant des constructions plus systematiques de ce genre de figures, dont la regularite est "invariante d'echelle".

II.3.4 Courbes fractales.

La plupart des courbes "fractales" tracees dans ce paragraphe ont ete inspirees par les tres belles illustrations du livre:

"The fractal geometry of Nature"
par B. Mandelbrot
(Freeman & Co, San Francisco, 1982)

La definition precise d'une courbe fractale est delicate et du ressort de la topologie et des mesures du plan affine reel. Notre vision sera d'autant plus terre a terre que la "topologie de l'ecran" est loin d'etre propice a une etude fine...

Nous ne decrirons jamais une courbe fractale elle-meme, mais une suite d'"approximants":

- au niveau 0: en general un segment.
- au niveau N: une suite finie d'approximants de niveau N-1.

Mandelbrot appelle le segment du niveau 0 "l'initiateur" (ce peut etre autre chose qu'un segment, par exemple un polygone ou un pave plein...), le passage du niveau 0 au niveau 1 (suffisant pour faire demarrer la recurrence) etant le "generateur".

1) La poussiere de Cantor. (Mandelbrot p. 80)

Mandelbrot dixit! il s'agit du celebre "ensemble triadique", non-denombrable et de mesure nulle. La resolution trop grossiere de l'ecran ne permet de visualiser que les tout premiers approximants:

```
? POUR CANTOR :N :C ; :N = niveau de recursion, :C = cote
> SI :N = 0 [AV :C STOP]
> CANTOR :N-1 :C/3
> LC AV :C/3 BC
> CANTOR :N-1 :C/3
> FIN
CANTOR EST DEFINI ; que son nom soit beni...
```

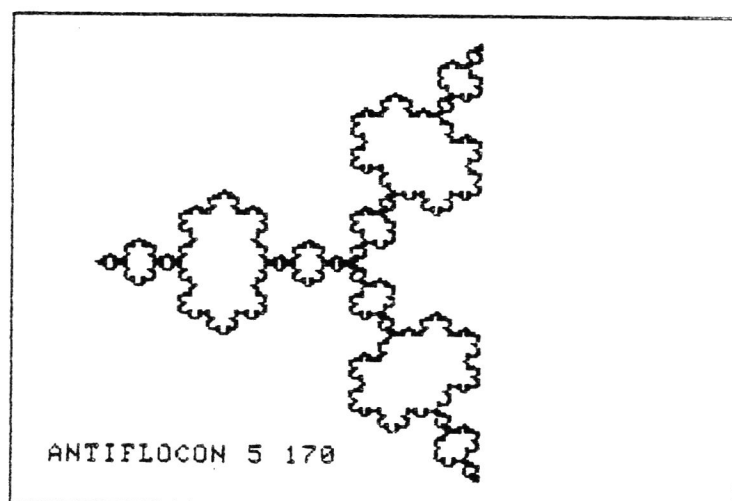
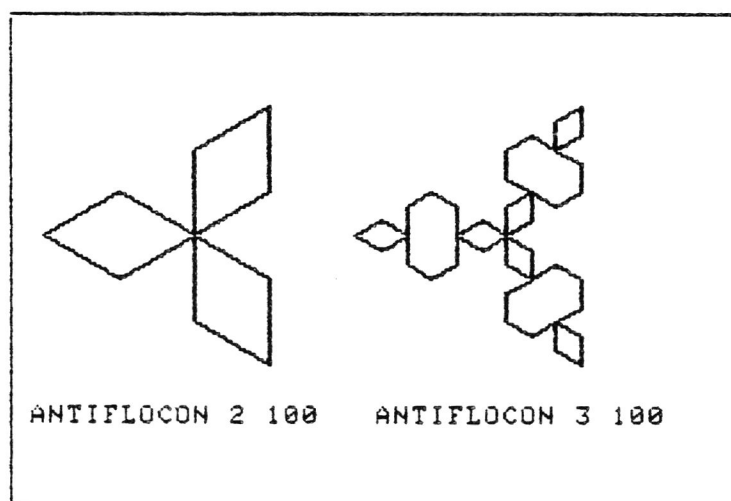
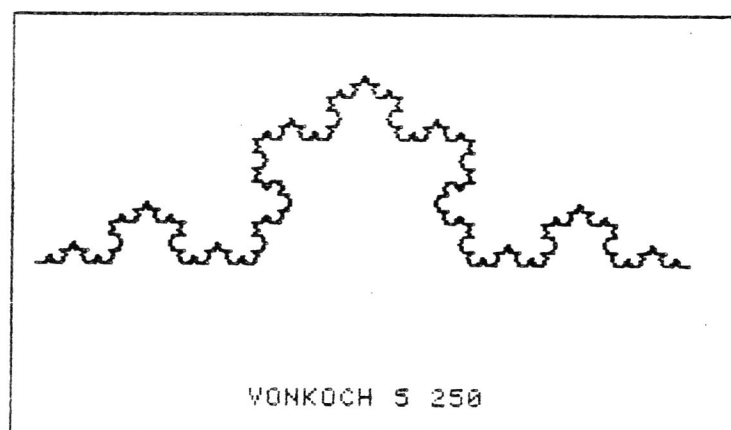
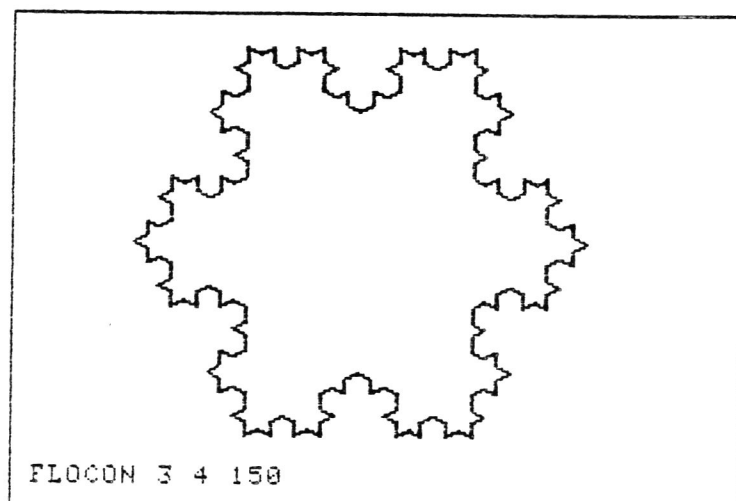
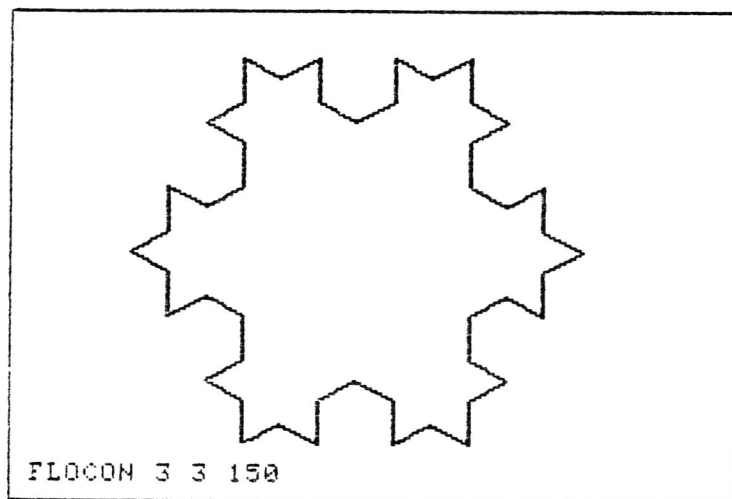
L'explication de la procedure reside dans le schema suivant de generation:

l'initiateur:



le generateur:





2) La flocon de neige. (Mandelbrot p.42)

Il s'agit de la Courbe de Von Koch. Nous la construirons sur les cotes d'un triangle equilateral; la generation sur chaque cote etant donnee par:



l'initiateur:



le generateur:

```

? POUR VON.KOCH :N :C
> SI :N = 0 [AV :C STOP]
> VON.KOCH :N-1 :C/3
> GA 60
> VON.KOCH :N-1 :C/3
> DR 120
> VON.KOCH :N-1 :C/3
> GA 60
> VON.KOCH :N-1 :C/3
> FIN
VON.KOCH EST DEFINI
?
? POUR POLY.VON.KOCH :NOMBRE :N :C
> REPETE :NOMBRE [VON.KOCH :N :C DR 360 / :N]
> FIN
POLY.VON.KOCH EST DEFINI
?
? POUR FLOCON :N :C
> POLY.VON.KOCH 3 :N :C
> FIN
FLOCON EST DEFINI

```

Pour l'eleve: Si l'on part d'un triangle equilateral de cote 1 et dont l'aire vaut donc $A = (\text{RAC } 3)/4$, montrer que la courbe limite "FLOCON ∞ 1" est de longueur infinie, mais delimite un domaine plan d'aire finie.

3) L'antiflocon.

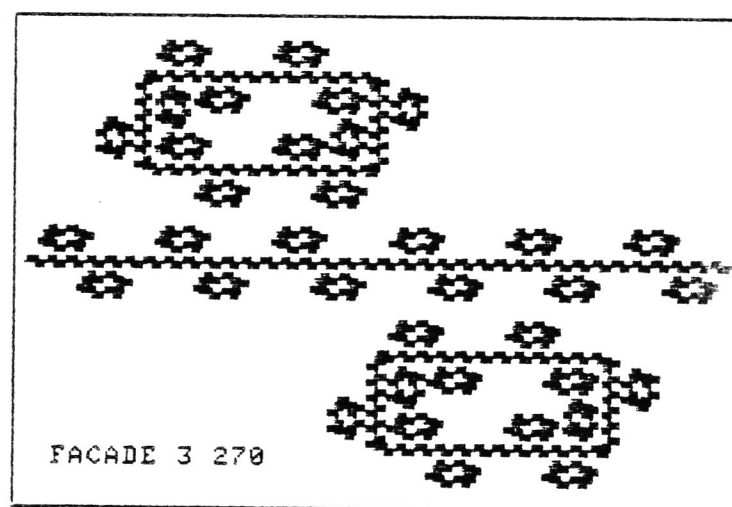
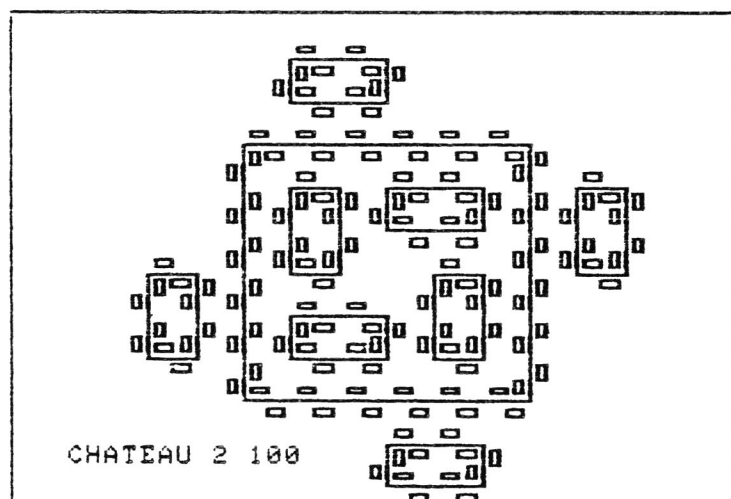
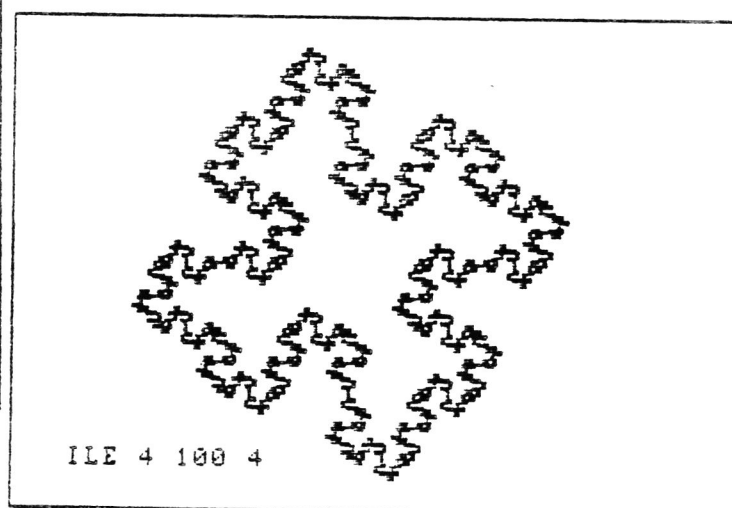
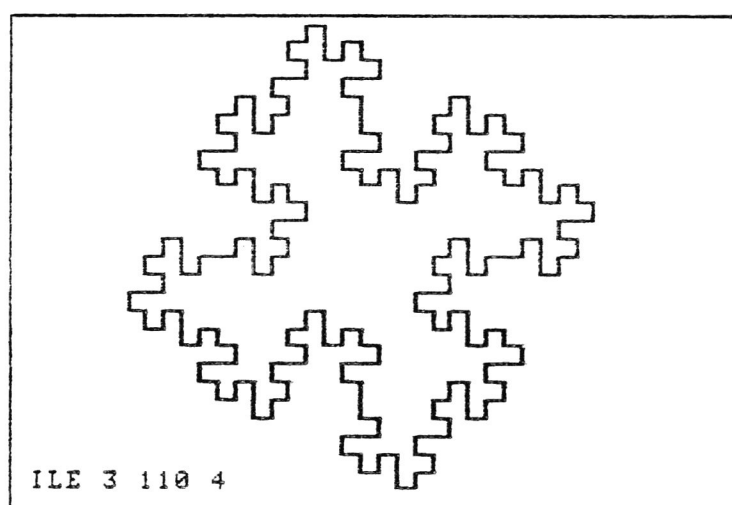
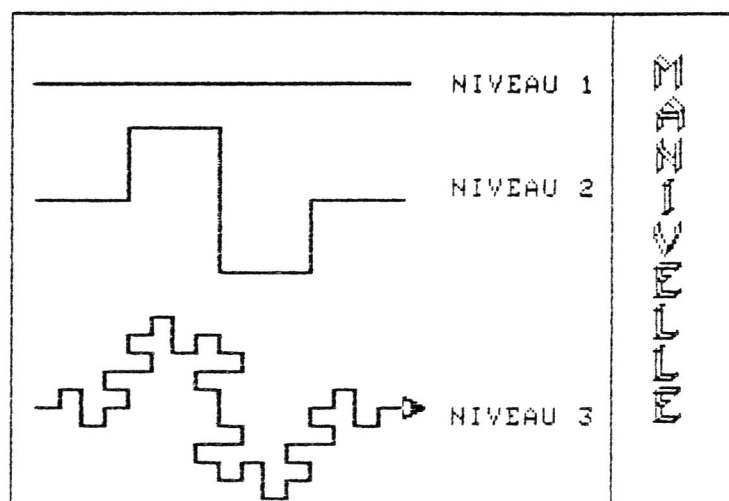
C'est une variante de la courbe de Von Koch, dans laquelle le "pic" est tourne en-dedans:

```

? POUR ANTI.FLOCON :N :C
> REPETE 3 [VON.KOCH :N :C GA 120]
> FIN
ANTI.FLOCON EST DEFINI

```

Pour l'eleve: reprise de l'exercice precedent; les resultats sont identiques. Que vaut la nouvelle aire ?

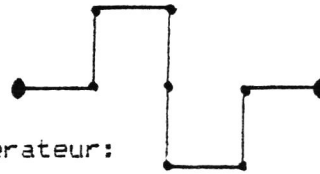


4) La manivelle. (Mandelbrot p.50)

l'initiateur:



le generateur:



```
? POUR MANIVELLE : C
> SI :N = 0 [AV :C STOP]
> MANIVELLE :N-1 :C/4
> GA 90
> REPETE 3 [MANIVELLE :N-1 :C/4 DR 90]
> GA 90
> REPETE 3 [MANIVELLE :N-1 :C/4 GA 90]
> DR 180
> MANIVELLE :N-1 :C/4
> FIN
MANIVELLE EST DEFINI
?
? POUR ILE :N :C :NOMBRE ; quel rapport avec une manivelle ?
> REPETE :NOMBRE [MANIVELLE :N :C DR 360 / :N]
> FIN
ILE EST DEFINI
```

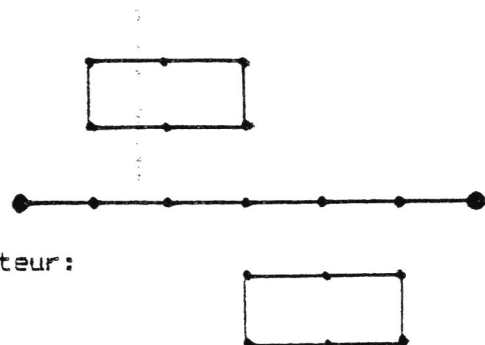
5) Le chateau. (Mandelbrot p.121)

Pour engendrer une "facade":

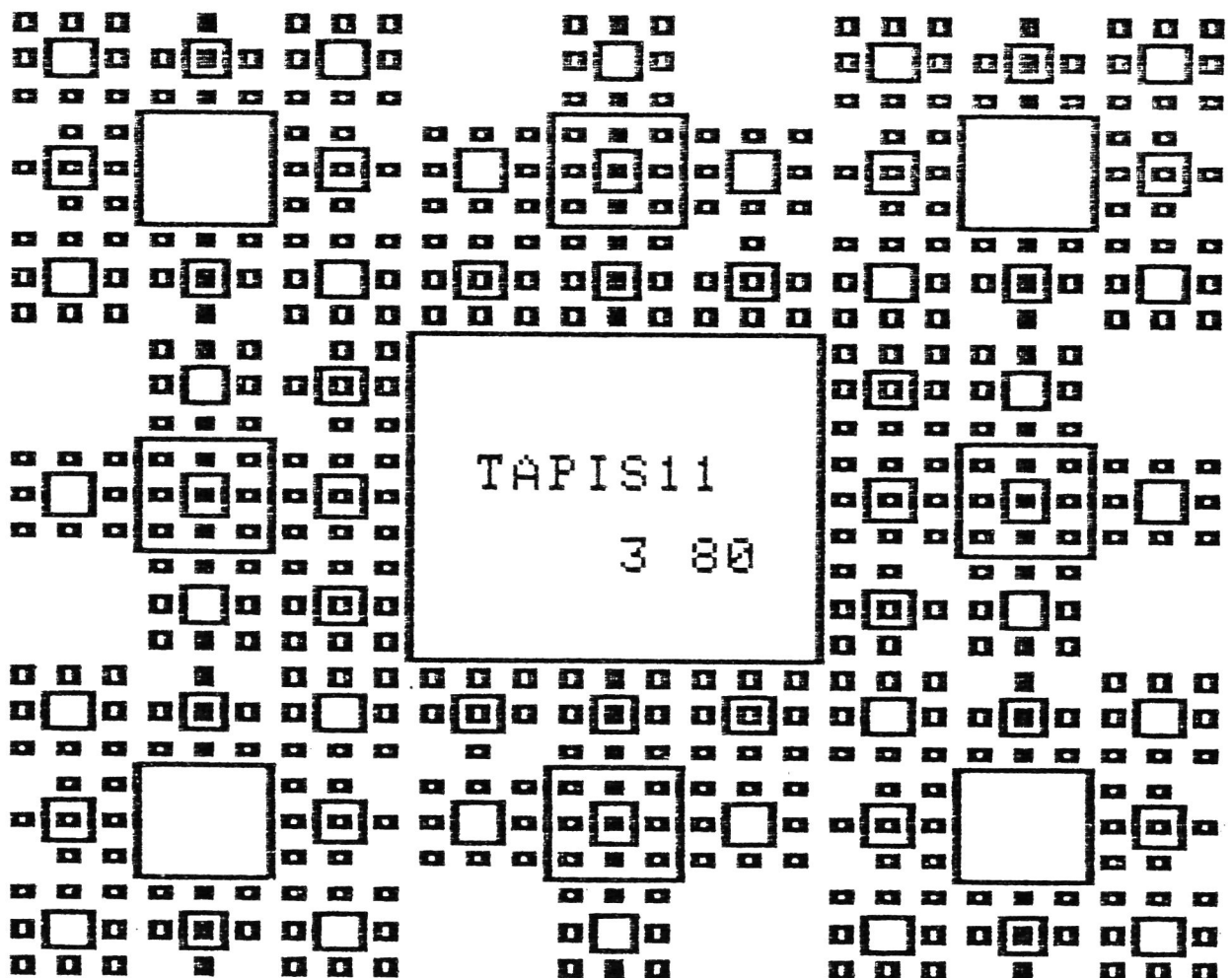
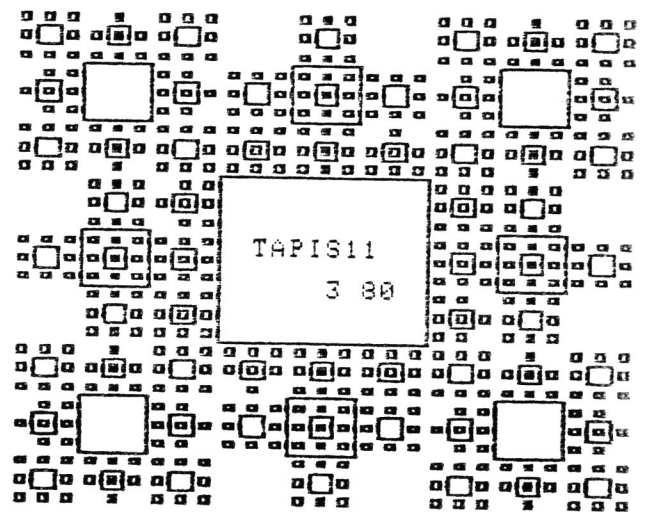
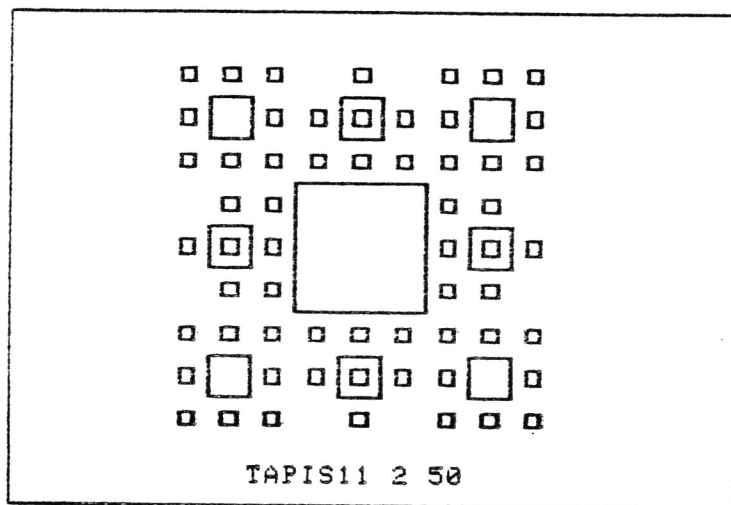
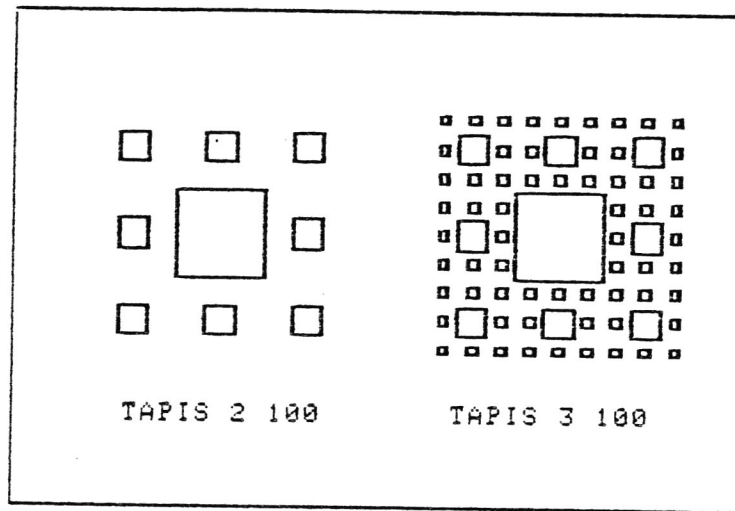
l'initiateur:



le generateur:



```
? POUR CHATEAU :N :C
> REPETE 4 [FACADE :N :C DR 90]
> FIN
CHATEAU EST DEFINI
?
? POUR FACADE :N :C
> SI :N = 0 [AV :C STOP]
> LOCAL "C1" DONNE "C1 :C/6
> REPETE 6 [FACADE :N-1 :C1]
> LC RE 5*:C1 GA 90 AV :C1 BC
> REPETE 2 [FACADE :N-1 :C1 DR 90 REPETE 2 [FACADE :N-1 :C1] DR 90]
> LC RE 2*:C1 DR 90 AV 2*:C1 BC
> REPETE 2 [REPETE 2 [FACADE :N-1 :C1] DR 90 FACADE :N-1 :C1 DR 90]
> LC AV 3*:C1 GA 90 AV :C1 DR 90 BC
> FIN
FACADE EST DEFINI
```



6) Le tapis de Sierpinski. (Mandelbrot p.144)

Il est un peu plus délicat à définir:

```
? POUR TAPIS :N :C
> LOCAL "C1 DONNE "C1 :C/3
> CARRE.CENTRAL :C
> SI :N = 0 [STOP]
> REPETE 2 [REPETE 3 [TAPIS :N-1 :C1 LC AV :C1 BC] DR 90 LC AV :C1 BC
  TAPIS :N-1 :C1 LC AV 2*:C1 BC DR 90]
> FIN
TAPIS EST DEFINI
?
? POUR CARRE.CENTRAL :C
> LC AV :C1 DR 90 AV :C1 GA 90 BC
> CARRE :C1
> LC RE :C1 DR 90 RE :C1 GA 90 BC
> FIN
CARRE.CENTRAL EST DEFINI
?
? POUR CARRE :C
> REPETE 4 [AV :C DR 90]
> FIN
CARRE EST DEFINI
```

N.B. Un ouvrage de 1^{ère} S (collection ISTR) propose un thème sur les objets fractals, notamment l'éponge de Menger (attribuée par les auteurs à Sierpinski), qui est le TAPIS en dimension 3.

Voici une autre version du TAPIS, moins symétrique, basée sur:



l'initiateur:  le generateur: 

l'initiateur étant ensuite répété sur chaque côté d'un carré:

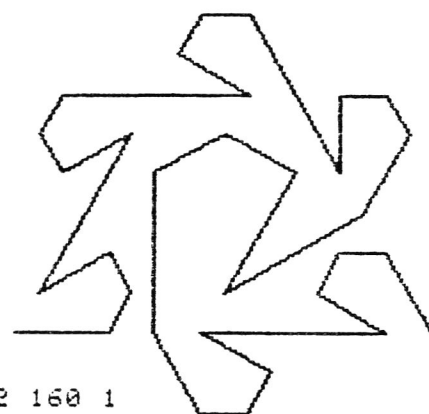
```
? POUR TAPIS11 :N :C
> REPETE 4 [TAPIS1 :N :C DR 90]
> FIN
TAPIS11 EST DEFINI
?
? POUR TAPIS1 :N :C
> SI :N = 0 [AV :C STOP]
> REPETE 3 [TAPIS1 :N-1 :C/3]
> LC RE 5*:C/3 GA 90 AV :C/3 BC
> REPETE 4 [TAPIS1 :N-1 :C/3 DR 90]
> LC DR 90 AV :C BC
> REPETE 4 [TAPIS1 :N-1 :C/3 GA 90]
> LC AV 2*:C/3 DR 90 AV :C/3 GA 90 BC
> FIN
TAPIS1 EST DEFINI
```



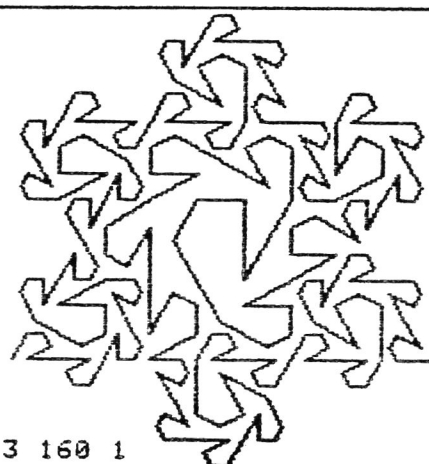
SINGE 1 100 1



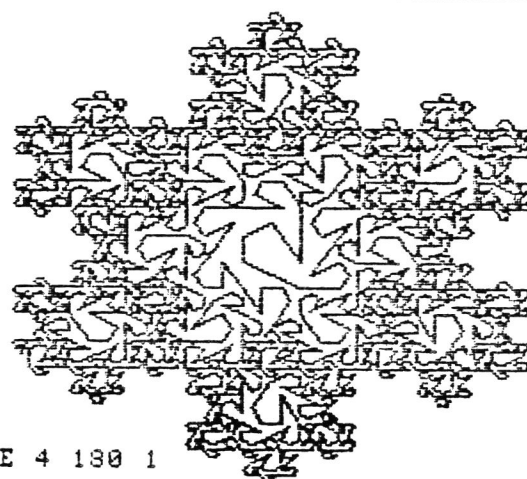
SINGE 1 100 -1



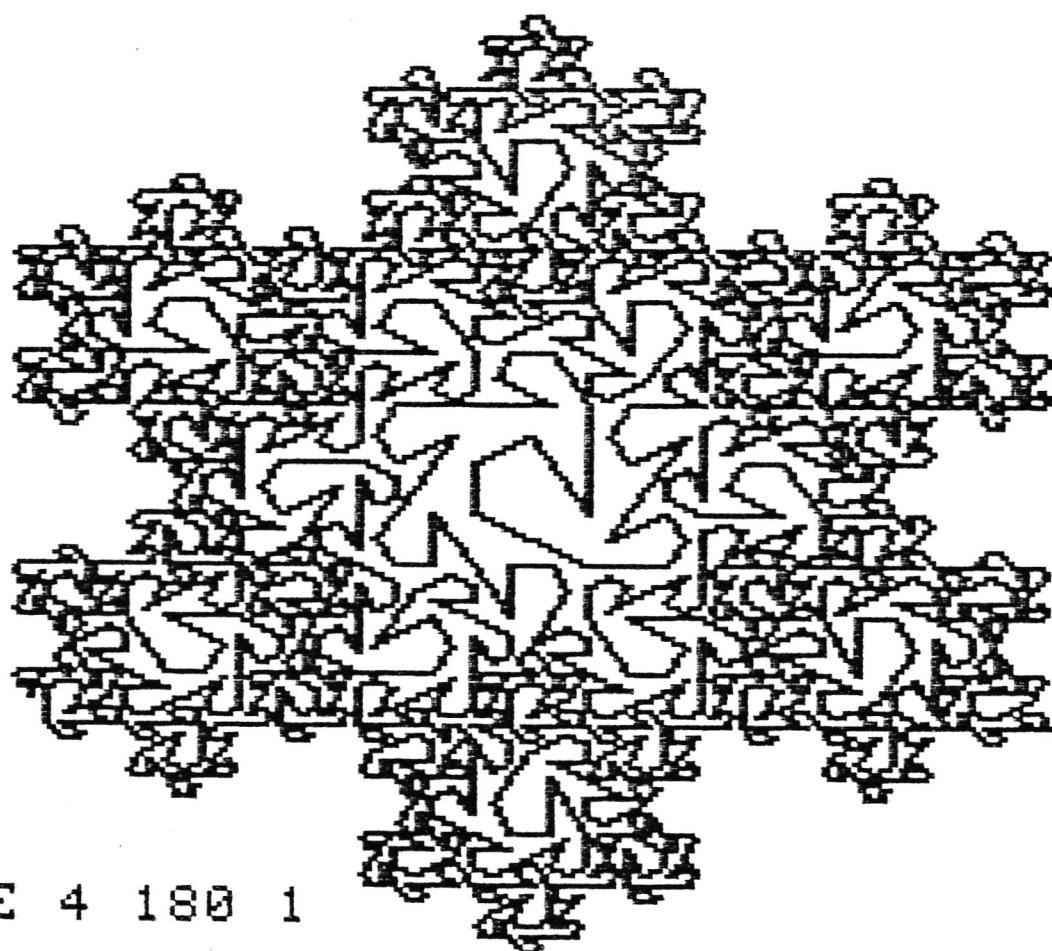
SINGE 2 160 1



SINGE 3 160 1



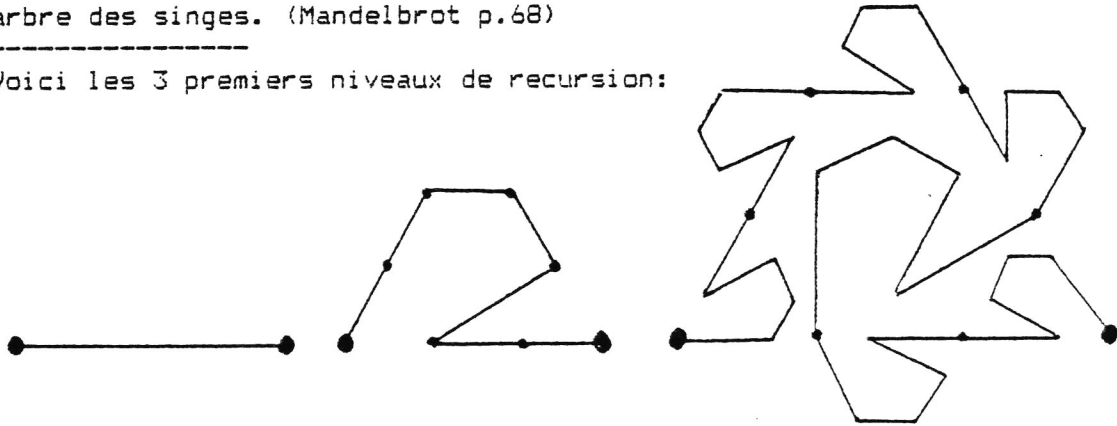
SINGE 4 130 1



SINGE 4 180 1

7) L'arbre des singes. (Mandelbrot p.68)

Voici les 3 premiers niveaux de recursion:

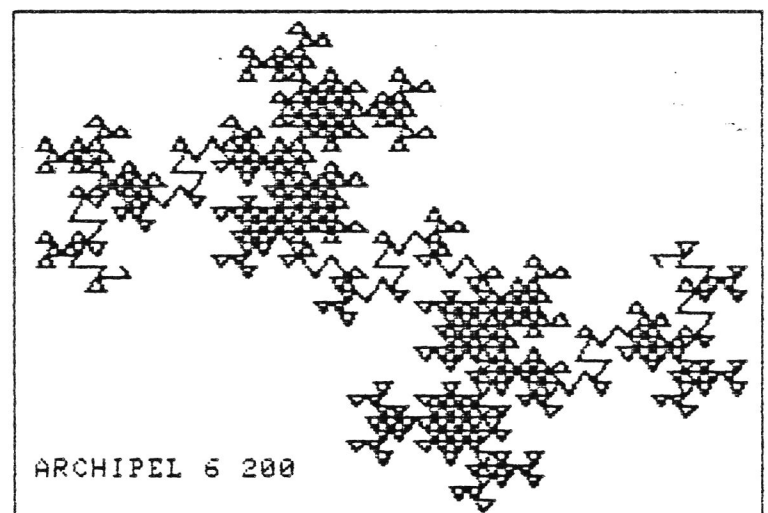


Le problème est que le "singe" de niveau N-1 peut être tracé à gauche ou à droite de chaque sous-segment du niveau N. Nous avons choisi de passer cette information sous la forme de 1 ou -1 (le paramètre DIR); le changement de direction s'effectue alors en changeant :DIR en -:DIR.

```
? POUR SINGE :N :C :DIR
> SI :N=0 [AV :C STOP]
> LOCAL "C1 DONNE "C1 :C/3 ; pour éviter des recalculs...
> GA :DIR*60
> SINGE :N-1 :C1 -:DIR
> LC AV :C1 DR 180 BC
> SINGE :N-1 :C1 -:DIR
> LC RE :C1 GA 120*:DIR AV :C1 DR 180 BC
> SINGE :N-1 :C1 -:DIR
> LC RE :C1 GA 120*:DIR AV :C1 DR 180 BC
> SINGE :N-1 :C1 -:DIR
> LC RE :C1 GA 90*:DIR AV :C1*RAC 3 DR 180 BC
> SINGE :N-1 :C1*RAC 3 :DIR
> LC RE :C1*RAC 3 DR 30*:DIR BC
> SINGE :N-1 :C1 -:DIR
> LC AV :C1 DR 180 BC
> SINGE :N-1 :C1 -:DIR
> LC RE :C1 DR 180 BC
> FIN
SINGE EST DEFINI
```

8) L'archipel.

```
? POUR ARCHIPEL :N :C
> SI :N = 0 [AV :C STOP]
> GA 60
> ARCHIPEL :N-1 :C/2
> DR 120
> ARCHIPEL :N-1 :C/2
> ARCHIPEL :N-1 :C/2
> GA 120
> ARCHIPEL :N-1 :C/2
> DR 60
> FIN
```

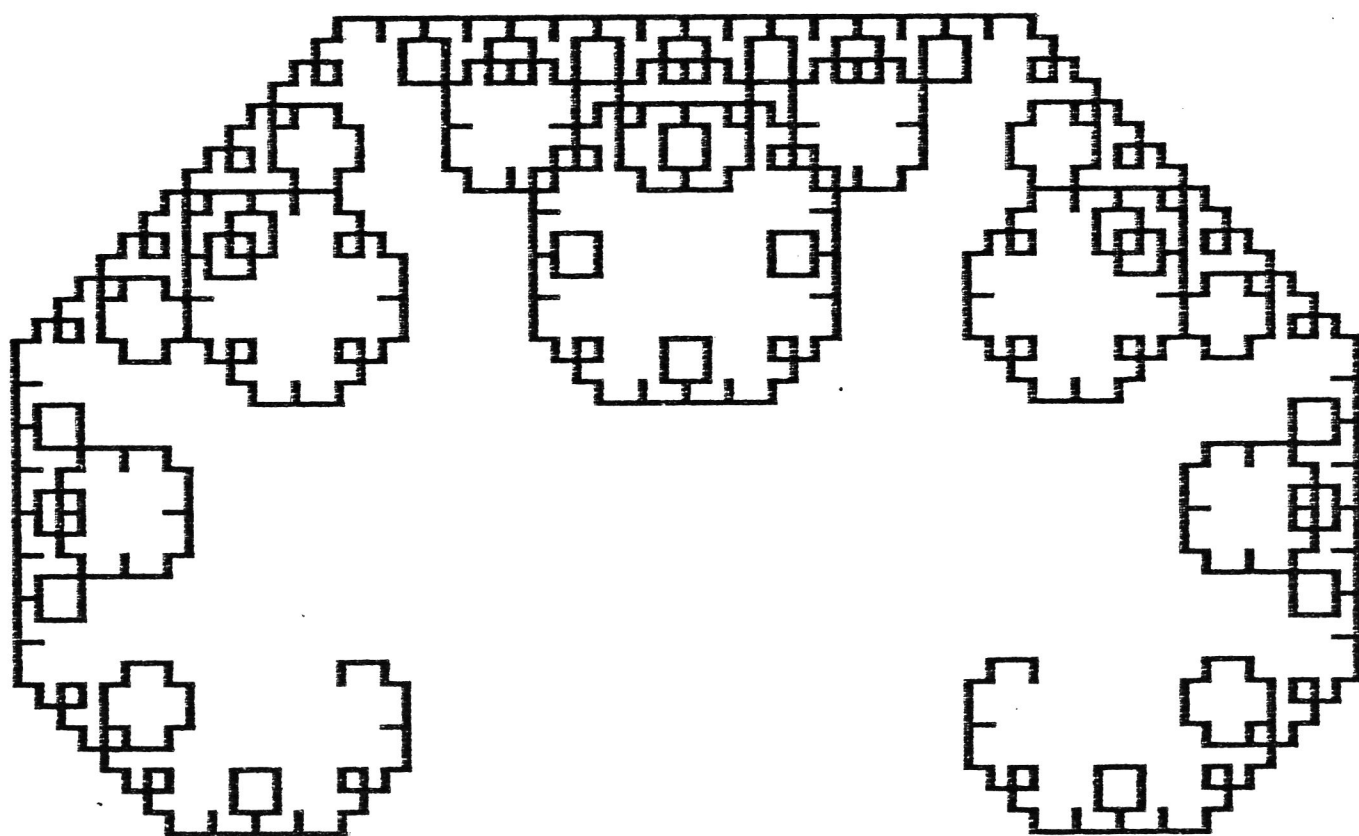
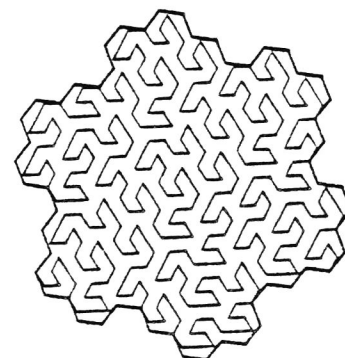
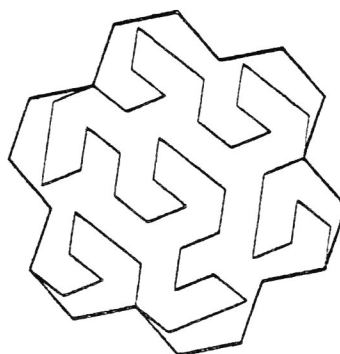
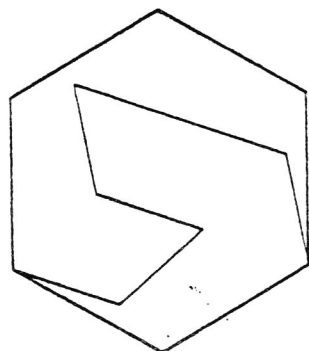


Decrivez vous-meme l'initiateur et le generateur !

Pour conclure sur les objets fractals, proposez aux élèves de réaliser un programme construisant la suite des approximations à partir de la donnée de l'initiateur et du générateur, soit sous forme de liste, soit de manière interactive avec un éditeur d'écran. L'exemple de l'arbre des singes, avec le problème des informations droite-gauche, montre qu'il ne faut néanmoins pas attendre un programme "trop" général...

Proposez aux élèves de construire le suivant, lui aussi classique.

Comment pourraient-ils engendrer la courbe "C" du second graphique ? (la solution est très simple...).



CC 10 4

. LOGO :

UN

LANGUAGE OUVERT.

III.1 Generalites sur LISP. Comparaison avec LOGO

Au debut etait LISP...Introduit vers 1958 par John Mac-Carthy [1] dans le but d'exploiter a fond la structure de "liste" (deja connue, par exemple grace aux travaux de Newell, Shaw et Simon [2] ou de Gelernter [3]), ainsi que l'idee generale de "lambda-calcul" developpee par le logicien Alonzo Church.

La nature interactive du langage etait l'un des objectifs de Mac-Carthy, qui militait fortement a l'epoque pour une programmation "interactive":

"...giving it a command, then seeing what happened, then giving it another command..."

et ceci pour des raisons liees aux applications envisagees, fortement axeées sur l'Intelligence Artificielle et sur la simulation du comportement humain.

Comme en LOGO, il n'y pas a proprement parler de notion d'"instruction" en LISP, tout au moins au sens que Von Neumann lui attribuait: un "operateur" agissant sur des donnees qui lui sont externes. A-propos de LISP, vous lirez souvent la formule lapidaire:

PROGRAMME = DONNEES

qui souligne l'identite structurelle totale entre les donnees (des S-expressions) et le "programme" (des S-expressions...). Ce que l'on appelle en LISP une S-expression n'est autre qu'un "objet" au sens de LOGO i.e. un "atome" (le "mot", numerique ou litteral, en LOGO) ou une "liste" (les crochets LOGO sont remplaces par des parentheses).

Voici quelques correspondances (la version LISP choisie est VLISP, de P. Greussay, qui est la plus portable sur micro-ordinateurs grace a l'implementation VLISP8 de J.Chailloux; elle est aussi porteuse d'idees recentes en LISP et nous la conseillons vivement):

- LOGO -

- (V)LISP -

1	? DONNE "X [1 2 3]	? (SETQ X '(1 2 3))
2	?	= (1 2 3)
3	? :X	? X
4	NE SAIS QUE FAIRE AVEC [1 2 3]	= (1 2 3)
5	? MONTRE :X	? (PRINT X)
6	[1 2 3]	(1 2 3)
7	?	= (1 2 3)
8	? DONNE "Y PR :X	? (SETQ Y (CAR X))
9	?	= 1
10	? DONNE "Z SP :X	? (SETQ Z (CDR X))
11	?	= (2 3)
12	? DONNE "X1 []	? (SETQ X1 NIL)
13	?	= NIL
14	? DONNE "X1 METF :Y :Z EC :X1	? (SETQ X1 (CONS Y Z))
15	[1 2 3]	= (1 2 3)
16	? EC EGALP :X :X1	? (EQUAL X X1)
17	VRAI	= T
18	? POUR FAC :N	? (DE FAC(N)
19	> SI :N=0 [RT 1] [RT :N * FAC :N-1]	(IF (EQ N 0)
20	> FIN	1
21	FAC EST DEFINI	(* N (FAC (1- N))))
22	?	= FAC
23	? EC (FAC 4) + 1	? (+ (FAC 4) 1)
24	25	= 25

```

25      ? POUR NEXTL :NOM                ? (NEXTL X) ; primitive;
26      > LOCAL "PREM                    = 1           ; comme CAR...
27      > DONNE "PREM PREMIER CHOSE :NOM ? X
28      > DONNE :NOM SP CHOSE :NOM      = (2 3)       ; (...presque !)
29      > RT :PREM
30      > FIN
31      NEXTL EST DEFINI
32      ? EC NEXTL "X
33      1
34      ? MONTRE :X
35      [2 3]

```

Avez-vous une première idée de la programmation en LISP ? Examinons à la lumière de ces quelques lignes les distinctions essentielles; car il y en a :

III.1.1 au moins une mineure (ligne 21 par exemple):

le deluge des parenthesés en LISP...

Question d'esthétique? Contraignante pour l'élève, elle contribue souvent à restituer la logique d'un corps de fonction compliqué, notamment en présence d'un grand nombre d'arguments avec appels imbriqués.

III.1.2 une autre plus importante:

il n'y a en LISP que des FONCTIONS

Alors que LOGO fait la distinction entre "procédure" et "fonction" (cette dernière étant une procédure retournant une valeur, comme en LSE), la programmation en LISP n'est autre qu'une suite d'appels d'"évaluation" d'une fonction (le CAR de la S-expression formant l'appel) à une liste d'arguments (le CDR de l'appel); en conséquence de quoi, cet appel va effectivement retourner une valeur, pouvant bien sûr être NIL (la liste vide, le "faux").

On peut objecter que certains appels, comme l'affectation ou la définition d'une fonction, n'ont pas à retourner une valeur, mais à effectuer un "effet de bord" sur la mémoire (i.e. une modification...). Telle est l'optique de LOGO: après DONNE à la ligne 1, le prompt ? apparaît aussitôt, en attente. LISP, en grand puriste, retourne bien une valeur, même à la ligne 22, après une définition de fonction (le nom de cette fonction définie: un atome littéral).

L'interprète LISP consiste donc en une super-fonction EVAL à qui on fournit une S-expression et qui l'évalue en l'interprétant comme un appel fonctionnel, sauf s'il s'agit:

- d'un atome numérique, qui est lui-même sa propre valeur;
- d'un atome littéral, qui admet comme valeur une S-expression;

C'est à cause de ce fonctionnement original de l'interprète que les langages de la famille LISP sont dits "applicatifs". LOGO en fait partie, même si les valeurs retournées ne sont pas effectivement imprimées pour des raisons qui sont plus pédagogiques et simplificatrices que profondes. (voir III.1.2).

Les successeurs de LISP et LOGO (au premier rang desquels figure le langage SMALLTALK) devraient avoir des capacités étonnantes, mais les constructeurs se font tirer l'oreille pour produire autre chose que des prototypes ou un matériel hors de prix (les "machines LISP" aux alentours de \$60,000; une machine LISP française va être produite sous peu à Toulouse). Il reste cependant possible, pour ceux que cela intéresse, d'obtenir à peu de frais un "micro-SMALLTALK" écrit en code VLISP [4,5].

LISP lui-même gagnerait à être développé sur micro-ordinateurs ayant entre 48 et 128 K-octets de mémoire vive: il n'en existe pas à l'heure actuelle (février 83) possédant à la fois:

- le graphisme haute-résolution;
- les reels pour le calcul scientifique;
- une mémoire-utilisateur "confortable";
- un compilateur.

Ceci est possible, et gageons qu'un tel "produit" aurait un "marché" plus vaste qu'il n'y semblerait à prime abord.

N.B. Mai 83 : la dernière version de VLISP, baptisée "LELISP 80", est disponible sur les micro-ordinateurs de l'Éducation Nationale fonctionnant avec le micro-processeur Z80 (une version portable existe sous le système d'exploitation CP/M).

=====

=====

III.2

=====

- LOGO en LOGO -

=====

III.2.1 Simulation de primitives.

Nous allons regarder de plus pres les primitives "de base" du langage, et voir s'il est possible d'en preciser l'effet en tachant de les "simuler" par elles-memes:

1) Le "SI".

```
? POUR SI% :TEST :ALORS :SINON                ; les parametres sont
> SI EXECUTE :TEST [EXECUTE :ALORS] [EXECUTE :SINON] ; passes sous forme
> FIN                                           ; de listes, ce qui
SI% EST DEFINI                                ; explique le EXECUTE :TEST.
? SI% [2=3] [EC 1] [EC 2]
2
```

Donc le "SI" opere de la maniere suivante: suivant la valeur de verite du test :TEST, il y aura EXECUTION de la liste :ALORS ou de la liste :SINON.

En fait, les appels a EXECUTE dans le corps de la fonction precedente sont inutiles:

```
? POUR SI% :TEST :ALORS :SINON
> SI (EXECUTE :TEST) :ALORS :SINON
> FIN
SI% EST DEFINI                                ; et vive la circularite...
? SI% [2=3] [EC 1] [EC 2]
2                                              ; en effet:
? EXECUTE [EXECUTE [EC 1]]
1
```

Nous pourrions vouloir par exemple un "SI" qui trouve un :TEST vrai des qu'il ne vaut ni "FAUX" ni []. A cet effet:

```
? POUR SI% :TEST :ALORS :SINON
> SI OU EGALP :TEST "FAUX EGALP :TEST [] :SINON :ALORS
> FIN
SI% EST DEFINI
? DONNE "L [1 2 3]
? SI% :L [MONTRE SP :L] [EC "AIE..."]
[2 3]                                         ; ce qui permet de n'executer une action
                                              ; que si une liste n'est pas vide par ex.
```

2) Le "POUR".

LOGO possede deux primitives pour construire des procedures: le POUR qui est suffisamment connu, mais aussi DEFINIS qui permet de definir dynamiquement (i.e. au moment de l'execution) de nouvelles procedures, ou meme de modifier d'anciennes procedures.

Un exemple d'utilisation de DEFINIS:

```
? POUR POUR% :NOM :LARG :CORPS
> DEFINIS :NOM METF :LARG LISTE :CORPS
> EC PHRASE [MAINTENANT JE SAIS COMMENT] :NOM
> FIN
POUR% EST DEFINI
? POUR% "PLUS [A B] [RETOURNE :A + :B]
MAINTENANT JE SAIS COMMENT PLUS
? EC PLUS 3 5
8                                              ; la procedure PLUS est dorenavant en memoire;
?                                              ; et pour simuler son effacement ?...
```



```

? POUR EFFACE% :PROC
> DEFINIS :PROC LISTE [] PHRASE LISTE "ECRIS METD :PROC [IMPOSSIBLE DE]
  [RENVOIE "NIVEAUSUP]
> FIN.
EFFACE% EST DEFINI
? EFFACE% "PLUS
? EC 1 + PLUS 2 3          ; appel de PLUS%
IMPOSSIBLE DE PLUS
? MONTRE TEXTE "PLUS      ; et pourtant elle est encore la:
[[ ] [EC [IMPOSSIBLE DE] RENVOIE "NIVEAUSUP]

```

N.B. : RENVOIE "NIVEAUSUP est une instruction stoppant inconditionnellement toute procédure en cours, et ramenant à la "boucle-système" de LOGO (cf. III.2.2 pour plus de renseignements sur ce NIVEAUSUP (en anglais TOPLEVEL)).

```

On accède au "texte" d'une procédure à l'aide de la primitive TEXTE:
? POUR RECTANGLE :LONG :LARG
> VIDECRAN
> REPETE 2 [AV :LARG DR 90 AV :LONG DR 90]
> FIN
RECTANGLE EST DEFINI
? MONTRE TEXTE "RECTANGLE
[[LONG LARG] [VIDECRAN] [REPETE 2 [AV :LARG DR 90 AV :LONG DR 90]]]

```

N.B. : Remarquez l'absence des ":" dans la liste de paramètres [LONG LARG]; le passage de paramètres en LOGO se fait "par valeur":

```

? POUR ESSAI X Y
> EC :X + :Y
> FIN
ESSAI EST DEFINI
? EDITE "ESSAI
va donner sur l'éditeur LOGO:      POUR ESSAI :X :Y
                                     EC :X + :Y
                                     FIN

```

avec rétablissement automatique des : indiquant le passage par valeur.

Ayant ainsi accès au "corps" de la fonction RECTANGLE par l'intermédiaire de la primitive TEXTE:

```

? POUR CORPS :PROC
> RT SP TEXTE :PROC
> FIN
CORPS EST DEFINI
? MONTRE CORPS "RECTANGLE
[[VIDECRAN] [REPETE 2 [AV :LONG DR 90 AV :LARG DR 90]]]

```

qui montre que le corps de la procédure RECTANGLE est en fait une liste, on peut visualiser d'encore plus près le fonctionnement de l'appel:

```

? RECTANGLE 10 20

```

En effet, que va-t-il se passer ?

- 1) Le premier élément de la liste TEXTE "RECTANGLE est une liste de paramètres (ici LONG et LARG). L'interprète va donc aller chercher après le mot RECTANGLE les valeurs actuelles des paramètres en question (ici 10 et 20), va les "lier" aux paramètres formels LONG et LARG en substituant à chaque occurrence de :LONG (resp. :LARG) dans CORPS "RECTANGLE les valeurs actuelles 10 (resp. 20).
- 2) Le nouveau "corps" ainsi "concretise" par les valeurs numériques de ses paramètres va être passé à EXECUTE qui va, comme il est censé le faire, exécuter la liste de commandes que comporte ce corps.
- 3) À la fin de cette exécution, si le programme comprend par ailleurs des variables nommées LONG ou LARG, celles-ci retrouvent leurs précédentes valeurs.

N.B. Ne pas prendre 1) au pied de la lettre, comme le montre l'exemple:

```

1  ? POUR CONTREXEMPLE :A
2  > DONNE "A :A+1
3  > EC :A
4  > FIN
5  CONTREXEMPLE EST DEFINI
6  ? CONTREXEMPLE 5
6                                ; :A aurait du etre remplace par 5 a la
                                ; ligne 3 si 1) etait correct...

```

Le point 1) devrait donc plutot etre:

```

1)' a.substituer 10 a :LONG et 20 a :LARG dans la 1ere instruction de
CORPS "RECTANGLE;
b.executer cette instruction avec les parametres actualises;
c.recommencer avec SAUFPREMIER CORPS "RECTANGLE.

```

Il se trouve que maintenant, l'etape b. peut elle-meme contribuer a une nouvelle mise a jour des parametres. En fait, le fonctionnement reel de l'évaluateur est bien plus complique, notamment a cause des possibilites d'appels recursifs.

Projet : Ecrire une procedure LOGO qui va simuler cette boucle:

"substitution <---> execution"

de la maniere suivante:

```

? DONNE "L TEXTE "RECTANGLE
? EXEC :L [10 20]           ; est cense simuler l'appel RECTANGLE 10 20
                             ; sans utiliser la procedure RECTANGLE, mais
                             ; uniquement son "texte" :L dans lequel vont
                             ; s'operer les substitutions, les parametres
                             ; etant passes dans une liste (ici [10 20]).

```

3) Le "REPETE".

Exemple: un generateur de carres emboites:

```

? POUR CARREEMBOITES :N :COTE :INCR
> REPETE :N [REPETE 4 [AV :COTE DR 90] DONNE "COTE :COTE+:INCR]
> FIN
CARREEMBOITES EST DEFINI           ; exercice: version recursive !...
? CARREEMBOITES 10

```



; joli ?...

La syntaxe de REPETE est donc:

REPETE <nombre-de-fois> <liste-d'instructions>

dans lequel <nombre-de-fois> peut etre une expression calculee:

```

? REPETE SI 1=2 [1] [2] [EC "LOGO]
LOGO
LOGO
? POUR REPETE% :N :INSTR
> SI :N > 0 [EXECUTE :INSTR] [STOP]
> REPETE% :N - 1 :INSTR
> FIN
REPETE% EST DEFINI
? REPETE% 3 [TAPE "LOGO]
LOGOLOGOLOGO

```

Projet : Ecrire un "REPETE% :INSTR :JUSQUA" dans lequel :INSTR et :JUSQUA sont passes sous forme de listes, et qui execute :INSTR jusqu'a ce que le test d'arret :JUSQUA soit egal a "VRAI. Appliquez votre procedure de la maniere suivante:

? DONNE "L [1 2 3 4]

? REPETE% [MONTRE :L DONNE "L SP :L] [VIDEP :L] ; montrer L et tous ses
; "saufpremiers".

4) Deux "boucles".

Projet : Ecrire une procedure:

BOUCLE :VAR :INITIAL :FINAL :INSTR
qui soit equivalente a la classique boucle "For" du BASIC:
? BOUCLE "I 1 3 [TAPE "LOGO TAPE :I]
LOGO1LOGO2LOGO3

Ecrire une procedure:

TANTQUE :TEST :INSTR
executant la liste :INSTR tant que :TEST ne vaut pas "FAUX":
? DONNE "X 0
? TANTQUE [:X<5] [TAPE :X DONNE "X :X+1] EC []
01234
?

III.2.2 Modification de la boucle-systeme.

Malgre le titre un peu trop prometteur, nous n'allons pas "reellement" toucher a l'interprete LOGO, qui est ecrit en code machine, mais bien plutot essayer de "simuler" une telle modification aux yeux de l'utilisateur (attention au virus de la simulation si vous jouez avec ces langages...).

Le paragraphe precedent avait mis en exergue quelques differences entre LISP et LOGO:

? 1+2	? (+ 1 2)
NE SAIS QUE FAIRE AVEC 3	= 3
?	?

Supposons que pour telle ou telle raison (par ex. pour ecrire un poly de 1'IREM) nous nous fixions pour objectif de forcer la reponse 3 a la question ? 1+2 , autrement dit d'attrapper au vol le message d'erreur "NE SAIS...".

Il nous faudra modifier ce que l'on appelle la "boucle-systeme" qui controle le fonctionnement de l'interprete au plus haut niveau (en anglais, le TOPLEVEL). L'introduction sur LISP avait deja laisse entrevoir l'action de cette boucle-systeme: lire-evaluer-ecrire:

en LISP: (PRINT (EVAL (READ))) et en LOGO: MONTRE EXECUTE LISLISTE

A cet effet, nous definissons circulairement la boucle-systeme par:

? POUR TOPLEVEL	
> TAPE "?	; le prompt d'attente (modifiable!..)
> MONTRE EXECUTE LISLISTE	; le corps de la boucle-systeme
> TOPLEVEL	; et le serpent se mord la queue...
> FIN	
?	
? TOPLEVEL	; on essaie ?...
? 1	
1	; encourageant,
? 1+2	
3	; itou,
? CARRE 10	; la procedure CARRE...
NE SAIS PAS COMMENT CARRE	; ...n'existe pas encore. Normal; mais:

? 1

NE SAIS QUE FAIRE AVEC 1 ; retour a l'ancienne boucle-systeme. Rate!

Les ennuis proviennent de l'oubli pur et simple de détourner les messages d'erreur indésirables pour ne garder que les "bons"...Il existe a cet effet une fonction `ATTRAPE` (en anglais `CATCH`) dont le fonctionnement se lit dans la fonction suivante `TOP` qui va passer aussitôt la main a `TOPLEVEL`, mais a la condition expresse qu'a la moindre erreur, elle puisse reprendre le controle pour gerer l'erreur commise suivant les besoins de la simulation en cours (cf III.2.3 pour de plus amples details sur ces fonctions "d'échappement").

Dans le cas qui nous interesse ici , le seul message d'erreur que nous voulons eliminer a pour code 10 (voyez la notice de votre propre machine); les notices d'erreurs seront delivrees par leur code:

```
? POUR TOP
> ATTRAPE "ERREUR [TOPLEVEL] ; passe conditionnellement la main a TOPLEVEL
> DONNE "ERR PREMIER ERREUR ; ERREUR est une fonction sans arguments qui
> ; retourne une liste dont le 1er element est
> ; le code de la derniere erreur commise.
> SI :ERR = 10 [] [(EC "ERREUR# :ERR)] ; si l'interprete croit "ne pas
> TOP ; savoir que faire avec..", qu'il
> FIN ; n'y pense plus !
TOP EST DEFINI
? TOPLEVEL
? 1+2
3
? TRI 10 40
ERREUR# 35 ; ne sais pas comment TRI...
? 1
1 ; j'y suis, j'y boucle...
```

Encore une fois, il ne s'agit pas de remplacer le message "NE SAIS PAS.." par l'horrible "ERREUR# 35", mais de montrer comment se fait une possible gestion des erreurs en LOGO: l'utilisateur se trouve, avec ce langage, en situation ouverte, il peut en modifier l'aspect a sa guise comme on le fait avec une boule d'argile...

III.2.3 Les fonctions d'échappement en LOGO

Il existe deux fonctions un peu speciales, `ATTRAPE` et `RENVOIE`, qui permettent dans certains cas de sortir "intempestivement" d'une procedure en cours et de transferer le controle de la situation a l'interieur d'une autre procedure:

```
? POUR GUIDER :ANGLE ; guidage de la tortue en mode direct;
> ATTRAPE "CONTROLE [DEMANDER] ; appelle la procedure DEMANDER, mais si un
> ; RENVOIE "CONTROLE est rencontre, alors la
> ; ligne suivante de GUIDER reprend la main;
> GUIDER :ANGLE
> FIN
GUIDER EST DEFINI
?
? POUR DEMANDER
> AV LIREMARCHE
> DR :ANGLE
> FIN
DEMANDER EST DEFINI
?
```

```

? POUR LIREMARCHE
> LOCAL "CHEMIN
> TAPE [!\ ] ; \ pour prendre le caractere "blanc";
> DONNE "CHEMIN PREMIER LL ; on attend un nombre;
> SI NON NOMBREP :CHEMIN ; renvoyer alors...
  [ECRIS [UN NOMBRE SVP!] RENVOIE "CONTROLE] ; ...a GUIDER;
> RETOURNE :CHEMIN
> FIN
LIREMARCHE EST DEFINI
?

```

```

? GUIDER 30
! 30
! 12LOGO
UN NOMBRE SVP!
!

```

```

! 30

```

```

; ETC...

```

=====

Bibliographie du Chapitre III

=====

- [1] : Mac-Carthy, J.
"Recursive functions of symbolic expressions and their computation by

machine"

in CACM, vol.3, 1960
- [2] : Newell, A.; Shaw, J.C.; Simon, H.A.
"Empirical explorations with the Logic Theory Machine: a case history
in Heuristics" (reprise d'un article de 1956)
in Feigenbaum & Feldman:
 "Computers and Thought" (Mac-Graw Hill, New-York, 1963).
(notons pour la petite histoire que leur programme, qu'ils baptiserent
"The Logic Theorist", dont le but etait la demonstration automatique de
theoremes, trouva une demonstration plus elegante que celle donnee par
Whitehead et Russell du theoreme 2.85 des "Principia Mathematica"...)
- [3] : Gelernter, H.
"Empirical explorations of the geometry-theorem proving machine"
(reprise d'un article de 1959)
in Feigenbaum & Feldman, loc.cit.
(Developpe dans un "FORTRAN List Processing Language", le programme de
Gelernter fut ecrit pour effectuer des demonstrations automatiques de
theoremes de geometrie plane.)
- [4] : Cointe, P.
"A VLISP implementation of SMALLTALK 76" (Septembre 1982)
Publications du LITP, Univ. Paris 6, Institut de Programmation.
- [5] : Cointe, P.
"Une extension de VLISP vers les objets"
Universite Paris 8, 2 rue de la liberte, Saint-Denis (janvier 83).
[une description de la Tortue-LOGO comme objet d'une classe SMALLTALK]

Sur LISP, nous conseillons particulierement:

- "The little LISPer" par D.P.Friedman
Science Research Associates, Palo Alto, California, 1974.
(tres facile a lire, moins a se procurer...)
- "LISP: Langage d'un autre type" par C.Queinnec, Eyrolles, 1982.

(utilise plus ou moins la version VLISP, qui devrait etre a disposition
des Lycees sous la denomination LELISP-80).

Sur l'histoire de ces langages et leurs liens avec le petit monde de l'Intel-
ligence Artificielle, d'ou ils sont issus, lire le tres attachant:

- "Machines who think" de Pamela McCorduck,

Freeman & Co, San Francisco, 1979.

=====

III.3 Dialogues.

Que de professeurs essayent avec divers succès d'établir le dialogue avec leurs élèves, LOGO va aussi s'attaquer à cette difficile tâche...
Le programme qui suit réalise sur un mode dialogue est un exercice d'application qui se veut plus motivant pour la recherche des diviseurs d'un nombre.

```
? POUR PARFAIT
> (LOCAL "N "VAL)
> EC []
> EC [UN NOMBRE EST DIT PARFAIT S'IL EST EGAL A LA SOMME DE SES DIVISEURS
    AUTRES QUE LUI-MEME]
> EC []
> EC [- SI CETTE SOMME EST INFERIEURE , ON DIT QU'IL EST DEFICIENT]
> EC []
> EC [- SI CETTE SOMME EST SUPERIEURE , ON DIT QU'IL EST ABONDANT]
> EC []
> EC [.....NOTE CES DEFINITIONS SUR UNE FEUILLE DE PAPIER.]
> ATTENDS 1200 ; ATTENDS est une procedure qui "ne
> (EC [] []) ; fait rien" pendant 1200 unites...
> EC [CE PROGRAMME A POUR BUT DE TE FAIRE TROUVER SI UN NOMBRE ENTIER EST
    DEFICIENT ,ABONDANT OU PARFAIT.]
> EC []
> LABEL "CONTINUE ; seulement pour montrer que les
> TAPE [TAPE UN NOMBRE ENTIER :] ; etiquettes existent. A éviter...
> DONNE "N PREMIER LL
> EC []
> RECHER1
> RECHER2
> FINAL
> (EC [] []) ATTENDS 300
> EC [ TAPE 0 SI TU VEUX T'ARRETER SINON TAPE UN AUTRE CARACTERE ] EC []
> DONNE "VAL PREMIER LL
> SI :VAL = 0 [STOP] [VA "CONTINUE] ; saut inconditionnel !
> FIN
PARFAIT EST DEFINI
?
? POUR RECHER1
> (LOCAL "ND "NB )
> TAPE [CHERCHE SES DIVISEURS , PUIS TAPE LE NOMBRE DE SES DIVISEURS:]
> DONNE "ND COMPTE DIVIS :N
> DONNE "NB PREMIER LL EC []
> TEST :NB = :ND
> SIF [SI :NB<:ND [EC [TU AS OUBLIE DES DIVISEURS , VERIFIE TES CALCULS!]
    EC [] ATTENDS 300 RECHER1 STOP] [EC [CERTAINS DES NOMBRES QUE TU AS
    TROUVE NE SONT PAS DES DIVISEURS ,VERIFIE TES CALCULS!] EC[] ATTENDS 300
    RECHER1 STOP]]
> FIN
RECHER1 EST DEFINI
?
? POUR RECHER2
> (LOCAL "SOM "SV "S )
> EC []
> TAPE [CHERCHE , PUIS TAPE LA SOMME DES DIVISEURS DU NOMBRE AUTRE QUE
    LUI-MEME:]
> DONNE "SV SOMMER SD DIVIS :N 0
> DONNE "S PREMIER LL EC []
> TEST :S=:SV
> SIF [EC [C'EST FAUX , RECOMMENCE!] ATTENDS 300 RECHER2 STOP]
> SIV [EC []] C'EST JUSTE ! <<<< ATTENDS 300]
> FIN
RECHER2 EST DEFINI
```

```

? POUR SOMMER :L
> SI VIDEP :L [RT 0]
> RT (PREMIER :L) + SOMMER SP :L
> FIN
SOMMER EST DEFINI
?
? POUR FINAL
> LOCAL "REP
> EC []
> EC [- TAPE 1 SI LE NOMBRE EST DEFICIENT ]
> EC [- TAPE 2 SI LE NOMBRE EST PARFAIT ]
> EC [- TAPE 3 SI LE NOMBRE EST ABONDANT ]
> DONNE "REP PREMIER LL EC []
> SI ET :REP=1 :SV<:N [EC [C'EST BON!] STOP]
> SI ET :REP=2 :SV=:N [EC [C'EST BON!] STOP]
> SI ET :REP=3 :SV>:N [EC [C'EST BON!] STOP]
> EC [ TU AS COMMIS UNE ERREUR ,RECOMMENCE!]
> (EC [] []) ATTENDS 300
> FINAL
> FIN
FINAL EST DEFINI

```

[Il est nécessaire de charger les procédures arithmétiques permettant de trouver la liste des diviseurs d'un nombre, cf. II.2.2]

Ce programme ne pose pas de difficultés de conception : le plus difficile est souvent la gestion du curseur sur l'écran: attention à la présentation du texte...

Pour réussir un tel programme, l'ordinateur doit donner l'impression de suivre l'exécutant pas à pas comme le ferait un précepteur... Ceci peut se révéler un aide certain pour la pratique des exercices en-dehors de la classe

On pourrait sans difficulté adjoindre au programme un compteur d'erreurs qui permettrait de noter l'exécutant. Chacun choisira suivant sa pédagogie.

Ainsi LOGO même sans tortue peut rendre de bons et loyaux services !...

=====

Appendice 1

- LES POLYNOMES EPARPILLES -

Ce programme a pour but la manipulation de polynomes en LOGO. Il permet en effet de:

- entrer au clavier un polynome;
- effectuer additions, soustractions, multiplications, calcul en un point ou derivation d'un polynome;
- stocker des polynomes sur disque.

Pour des raisons tenant a la fois a l'encombrement-memoire et a la nature de LOGO, la structure de liste a ete choisie pour représenter un polynome. De plus, ce mode de representation est assez proche de celui a-travers lequel nous "pensons" un polynome: on ne tient aucun compte des coefficients nuls, au contraire de ce qui se fait usuellement avec des tableaux.

Un polynome est donc transforme en une liste de monomes, chaque monome etant forme d'un couple exposant-coefficient.

Exemples:

```
MONOMES:      1 -> exposant 0 coefficient 1 -> [0 1]
              X ->          1          1 -> [1 1]
              2X ->          1          2 -> [1 2]
              3X4 ->         4          3 -> [4 3]
```

```
POLYNOMES: X+1 -> [1 1 0 1]
            3X5-X2-4 -> [5 3 2 -1 0 -4]
            ce dernier exemple montre bien l'oubli des coefficients
            des termes de degres 3, 4 et 1. Ou encore:
            X100-1 -> [100 1 0 -1]
            comparez a l'écriture correspondante en tableau...
```

```
>>>>> Note de dernière minute: il ne semble pas utile de détailler la tren-
>>>> taine de procédures utilisées par l'élève pour son projet. Celui-ci
>>> "tourne" bien, et les détails vous sont laissés. Quelques précisions,
>>> par contre, sur l'originalité de la structure de données ici utilisée
>> pour représenter un polynome (code "un peu comme à la main").
> Comment se "pensent" les opérations algébriques usuelles entre poly-
> nomes ?
```

Le principe général, là encore, sera récursif: monome apres monome , on avance dans les listes codant les polynomes.

- ADDITION : l'addition se fait monome a monome, a condition qu'ils aient le meme exposant, et on n'additionne alors que les coefficients.
- PRODUIT : le produit general est connu des que l'on sait faire l'addition de deux polynomes, et la multiplication d'un polynome par un seul monome; cette dernière est facile: multiplication par le coefficient, et addition de l'exposant.
- DERIVEE : ne pose aucun probleme.
- VALEUR : le calcul de la valeur d'un polynome en un point donne :X s'effectue par un schema de Horner (procède bien sur récursif...):

$$2X^3 - X^2 + 3X - 2 = -2 + X(3 + X(-1 + 2X))$$
pour lequel il faut savoir "mettre X en facteur".

Un peu plus technique, nous donnons ci-dessous les procédures d'entrées-sorties écrites par l'élève pour sauver les polynômes sur disque. En fait, elles permettent plus généralement de sauver n'importe quelles listes passées par leurs noms.

```
Exemple:  ? DONNE "L1 [2 4 6] DONNE "L2 [1 3 5 7]
           ? SAUVEL "NOMBRES [L1 L2]           ; NOMBRES est le nom du fichier.
           ? DONNE "L1 [] DONNE "L2 []         ; gommage de L1 et L2.
           ? MONTRE :L2
           []                                   ; verification...
           ? RAMENEL "NOMBRES                   ; lecture sur disque du fichier.
           ? MONTRE :L2
           [1 3 5 7]
```

N.B. Ces procédures risquent de dépendre de la version choisie. Mais le principe de base restera le même si le besoin s'en fait sentir: lorsque LOGO n'accepte de ne sauver que des fonctions, il faut lui faire sauver des listes en le laissant croire qu'il s'agit d'une fonction (temporaire) !!

- Entrées-Sorties sur disque -
=====

```
1) appel:  SAUVEL "NOMFICHIER "LISTE
           ou SAUVEL "NOMFICHIER [LISTE1 LISTE2 ... LISTEn]
```

utilite: sauve dans le fichier disque NOMFICHIER la liste de nom LISTE (avec son nom), ou les listes de noms LISTE1,...,LISTEn (avec leurs noms).

```
POUR SAUVEL :NOMFICHIER :NOMLISTE
SI MOTP :NOMLISTE [DONNE "NOMLISTE LISTE :NOMLISTE]
DEFINIS "SAUVELISTES!!! LISTE [] SAUVEL1 :NOMLISTE
GROUPE "SAUVEL2 "SAUVELISTES!!!
SAUVE :NOMFICHIER "SAUVEL2
EFFACE "SAUVELISTES!!!
FIN
```

```
POUR SAUVEL1 :NOMLISTE
SI VIDEP :NOMLISTE [RT []]
RT (PH "DONNE LISTE MOT ""
      PREMIER :NOMLISTE
      SAUVEL2 CHOSE PREMIER :NOMLISTE
      SAUVEL1 SP :NOMLISTE)]
FIN
```

```
POUR SAUVEL2 :NOMLISTE
SI MOTP :NOMLISTE [RT MOT "" :NOMLISTE] [RT :NOMLISTE]
FIN
```

```
2) appel:  RAMENEL "NOMFICHIER
```

fonction: ramene les listes (avec leurs noms), sauvees dans le fichier disque NOMFICHIER.

```
POUR RAMENEL :NOMFICHIER
RAMENE :NOMFICHIER
SI NON DEFINIP "SAUVELISTES!!!
  [EC [# CE N'ETAIT PAS UN FICHIER DE LISTES...] STOP]
SAUVELISTES!!!
EFFACE "SAUVELISTES!!!
FIN
```

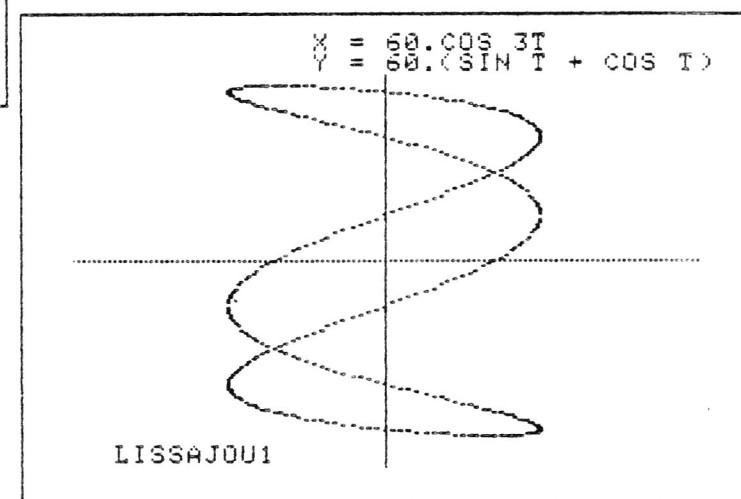
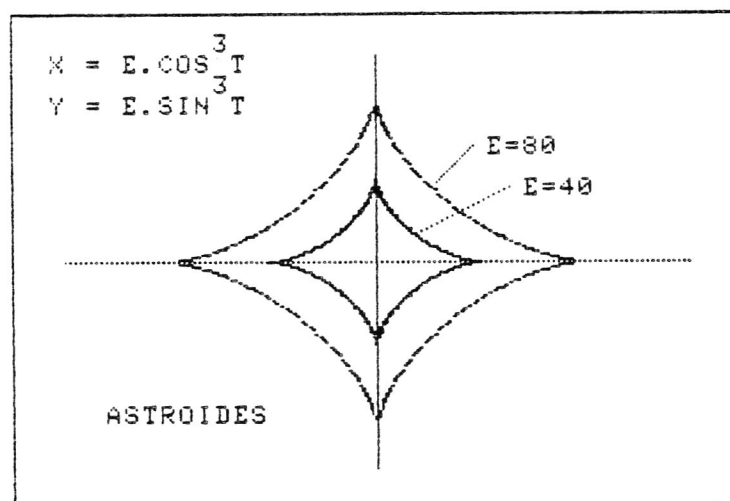
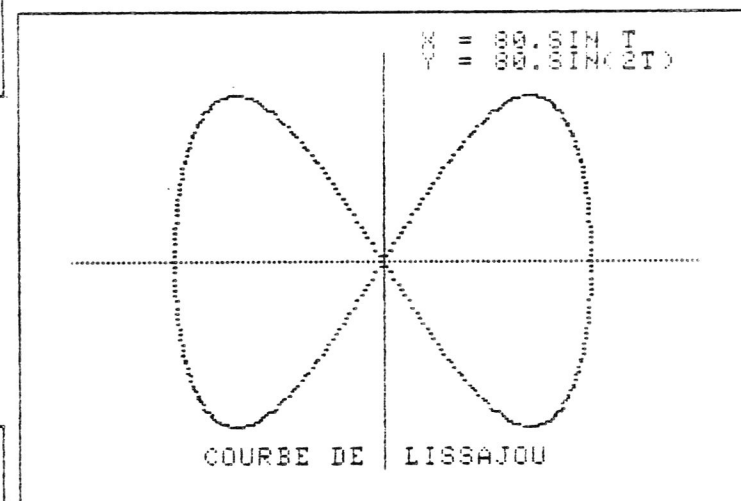
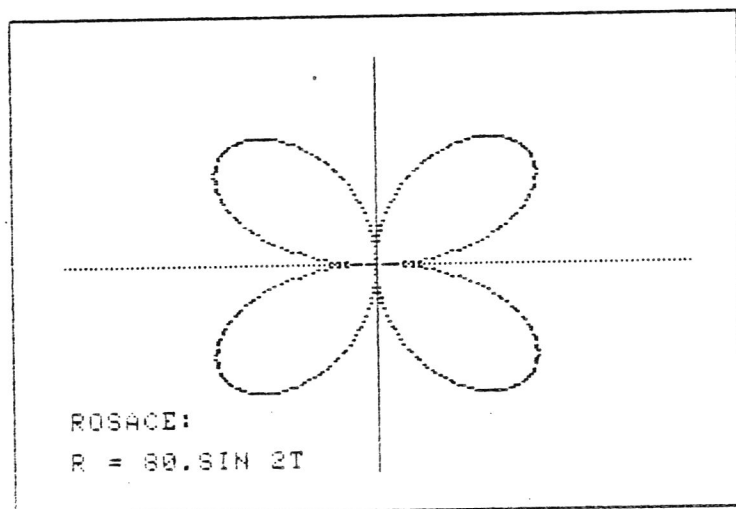
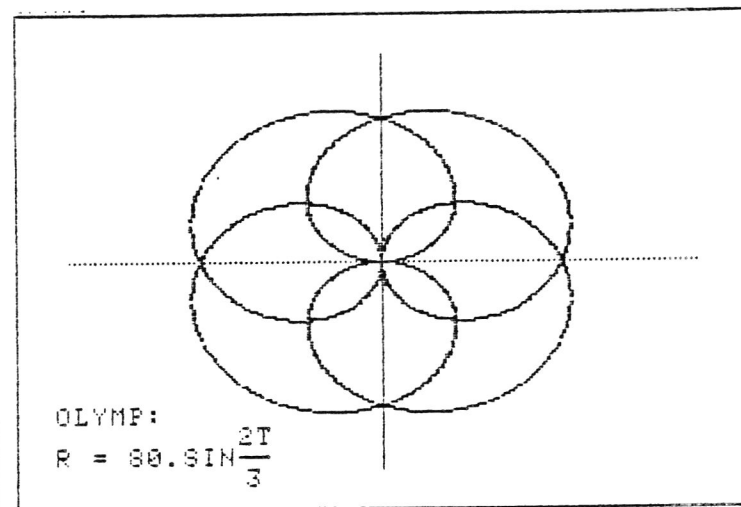
=====

Appendice 2

- TRACES DE COURBES -

Le programme qui suit a pour but de tracer des courbes définies paramétriquement, en coordonnées polaires ou cartésiennes, les équations de ces courbes étant fournies par l'utilisateur lui-même.

```
? POUR COURB
> (LOCAL "RP "P "REP "VX "VY "H "K )
> VIDETEXTE TEXTECRAN
> EC [CE PROGRAMME PERMET DE TRACER DES COURBES DE FONCTIONS DONNEES EN
    COORDONNEES CARTESIENNES OU DES FONCTIONS PERIODIQUES ,DEFINIES EN
    POLAIRES OU PARAMETRIQUEMENT] EC []
> EC [TAPE Ø SI LA FONCTION EST DONNEE EN COORDONNEES CARTESIENNES ,SINON
    TAPE UNE AUTRE LETTRE ]
> DONNE "RP PREMIER LL EC []
> SI :RP=Ø [CARTESIEN STOP]
> TAPE [TAPE LA PERIODE DE LA FONCTION EN DEGRES:]
> DONNE "P PREMIER LL
> EC [TAPE 1 SI LA COURBE EST DEFINIE EN POLAIRE SINON TAPE UNE AUTRE
    LETTRE]
> DONNE "REP PREMIER LL EC [] EC []
> SI :REP=1 [POLAIRE STOP]
> EC [TAPE L'EQUATION EN "T" DE LA VARIABLE X :]
> TAPE [X=]
> DONNE "VX LL
> DONNE "H [DONNE "X EXECUTE :VX] EC []
> EC [TAPE L'EQUATION EN "T" DE LA VARIABLE Y :]
> TAPE [Y=]
> DONNE "VY LL
> DONNE "K [DONNE "Y EXECUTE :VY] EC[]
> AXE
> TRACER Ø
> FIN
COURB EST DEFINI
?
? POUR CARTESIEN
> LOCAL "L "H "V "X1 "X2 "T )
> EC [TAPE L'EQUATION EN :X DE LA FONCTION Y] EC []
> TAPE [Y=]
> DONNE "L LL
> DONNE "H [DONNE "Y EXECUTE :L]
> EC [ENTRE QUELLES VALEURS DE X DESIRES-TU LE TRACE DE LA COURBE]
> DONNE "V LL EC []
> DONNE "X1 PREMIER :V
> DONNE "X2 DERNIER :V
> EC [QUEL ACCROISSEMENT DE X VEUX-TU A CHAQUE FOIS?]
> DONNE "T LL EC []
> AXE
> EXEC :X1
> FIN
CARTESIEN EST DEFINI
?
? POUR AXE
> CENTRE CT FCC 3
> FX -120 FX 120
> CENTRE
> FY -100 FY 100
> FCC 1
> FIN
AXE EST DEFINI
```



```

? POUR EXEC :X
> SI :X>:X2 [STOP]
> EXECUTE :H
> SI ET ABS :X<120 ABS :Y<90 [POINT LISTE :X :Y]
> EXEC :X+PREMIER :T
> FIN
EXEC EST DEFINI
?
? POUR ABS :X
> SI :X>0 [RT :X] [RT -:X]
-> FIN
ABS EST DEFINI
?
? POUR POLAIRE
> LOCAL "L
> EC [TAPE L'EQUATION EN "T" DE LA COURBE EN COORDONNEE POLAIRE] EC []
> TAPE [R=]
> DONNE "L LL
> DONNE "L METF "(" :L
> DONNE "L METD ")" :L
> DONNE "H [DONNE "X EXECUTE PH :L [*COS :T]]
> DONNE "K [DONNE "Y EXECUTE PH :L [*SIN :T]]
> EC []
> AXE
> TRACER 0
> FIN
POLAIRE EST DEFINI
?
? POUR TRACER :T
> DONNE "T :T
> EXECUTE :H EXECUTE :K
> SI ET ABS :X<120 ABS :Y<90 [POINT LISTE :X :Y]
> SI :T=:P [STOP]
> TRACER :T+1
> FIN
TRACER EST DEFINI

```

La seule difficulté de ce programme est l'interprétation des expressions de la forme DONNE "H [DONNE "X EXECUTE :VX].

Il faut savoir que l'instruction EXECUTE retourne ce que l'opération :VX retourne et lorsque l'on trouve EXECUTE :H, LOGO exécutera la liste d'instructions :H.

L'utilisateur devra se méfier lorsqu'on lui demandera une fonction, celle-ci devra retourner une valeur.

Ainsi après : TAPE L'EQUATION EN X DE LA VARIABLE Y on devra écrire par exemple: :X*:X-3 si l'on veut que LOGO y retrouve ses petits.

=====

Appendice 3 - SUR UNE REPARTITION DES NOMBRES PREMIERS -

Voici les deux procédures qui vont nous permettre de savoir si un nombre entier est premier:

```

? POUR PREM :X                    ; :X > 1
> SI :X < 4 [RT "VRAI]
> SI OU MULTIPLE :X 2 MULTIPLE :X 3 [RT "FAUX] [RT PREM :X 1]
> FIN
PREM EST DEFINI
?
? POUR PREM1 :X :N
> SI :X < (6*:N-1) * (6*:N-1) [RT "VRAI]
> SI OU MULTIPLE :X 6*:N-1 MULTIPLE :X 6*:N+1 [RT "FAUX]
> RT PREM1 :X :N+1
> FIN
PREM1 EST DEFINI
?
? POUR MULTIPLE :X :Y
> RT RESTE :X :Y = 0
> FIN
MULTIPLE EST DEFINI

```

La procédure PREM s'occupe de la divisibilité par 2 et 3 et, si elle n'a pu conclure, elle passe la main à PREM1, qui sait que les nombres premiers (sauf les...premiers venus 2 et 3) se cachent derrière les entiers de la forme $6n-1$ ou $6n+1$.

Comme application tortueuse (due à S.M. Ulam, en 1963, cf. "Jeux Mathématiques du Scientific American", Cedic ed.), visualisons à l'aide d'une spirale la répartition de la suite des nombres premiers, disons jusqu'à 30000.

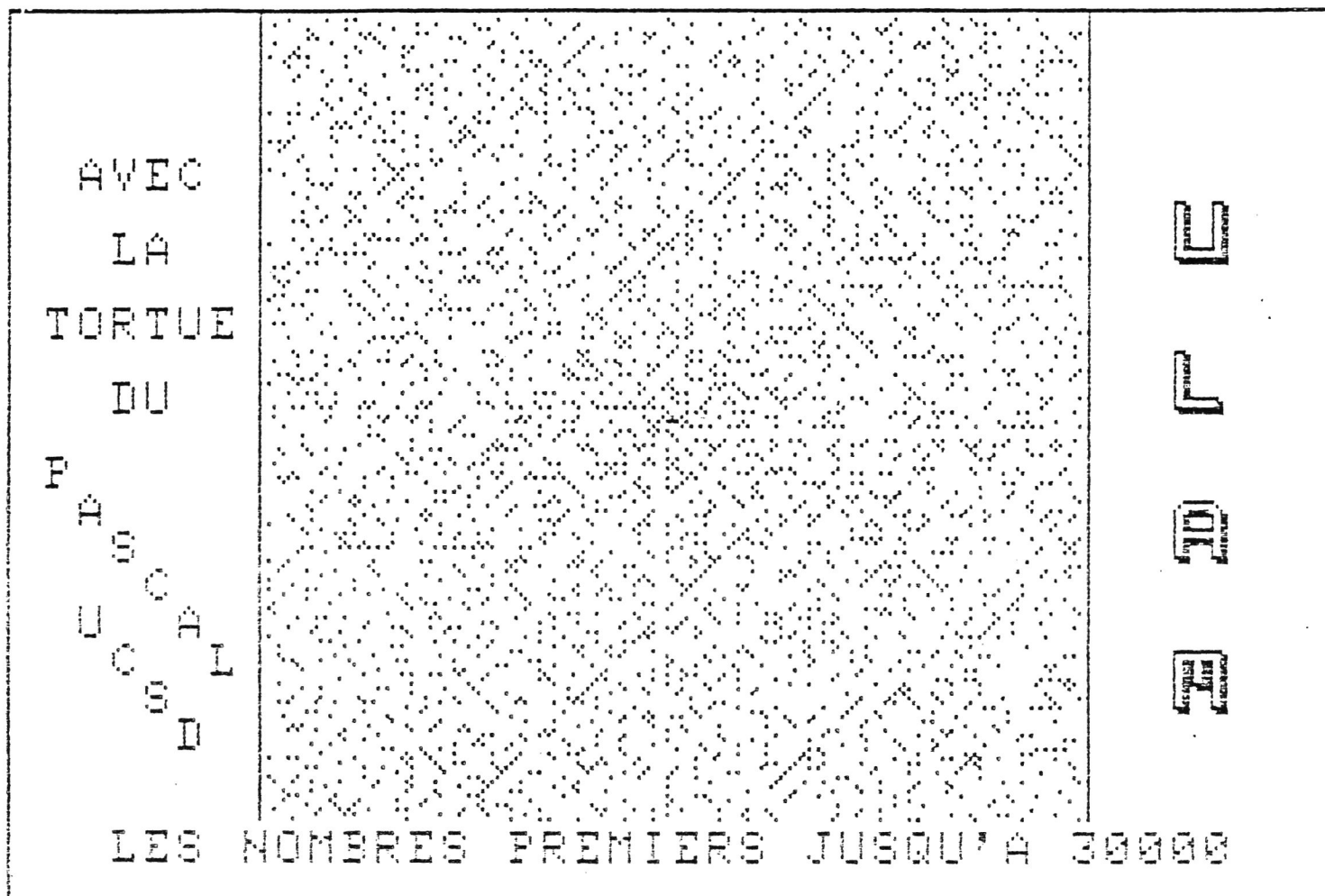
La tortue parcourt une spirale représentant la suite des nombres entiers. Une trace sous forme de point est laissée par la Tortue à chaque nombre premier. Les deux procédures suivantes mettent en jeu cette idée:

```

? POUR ULAM
> LOCAL "X
> DONNE "X 2                    ; X représente l'entier courant.
> VE CT LC
> VISU 1
> FIN
ULAM EST DEFINI
?
? POUR VISU :N
> SI :N > 30000 [STOP]
> REPETE 2 [REPETE :N [DONNE "X :X+1 AV 1 SI PREM :X [BC POINT POS LC]]
  GA 90]
> VISU :N+1
> FIN
VISU EST DEFINI

```

L'APPLE II ayant des problèmes au niveau de la définition graphique en LOGO, nous vous laissons implémenter ces deux procédures sur votre propre machine. Le dessin ci-contre a été obtenu à partir d'un programme analogue en Pascal-UCSD, utilisant les mêmes primitives graphiques que la Tortue LOGO.



En fait, il serait bon de "lire" sur un fichier la liste toute prête des nombres premiers < 30000 . C'est ce que nous avons fait avec le programme Pascal. Il est à regretter que - au moins dans notre version - les entrées-sorties vers la disquette soient peu évoluées, et lentes. Ceci, joint à une mémoire disponible étroite, fait que ce conseil risque fort de rester un vœu pieux...

N°	Intitulé
4	GROUPE FRANCAIS/MATHEMATIQUES (Tome 1)
10	CALCUL MENTAL
11	CALCUL PREMIER CYCLE
14	DE LA TEMPERATURE RESULTANTE SECHE A L'ANGLE SOLIDE
15	GROUPE FRANCAIS/MATHEMATIQUES (Tome 2)
16	LES JEUX DU CLUB DES CORDELIERS
21	ALMANATHEMATIQUES
22	FRISES
24	CLIMATisons les mathematiques
25	les forces en statique
26	BIBLIOGRAPHIE POUR L'ENSEIGNEMENT DES MATHEMATIQUES
28	RELATIONS/ANGLES
29	LA VITESSE
31	FONCTIONS ET REPRESENTATIONS GRAPHIQUES
32	LES ANGLES
34	METROLOGIE
35	ACTIVITES SPACIALES
36	TRAINS EPICYCLOIDIAUX
38	CONCEPTION DU CERCLE CHEZ ELEVES DE L'ECOLE ELEMENTAIRE
39	POT-POURRI
40	GROUPE DE RECHERCHE AVIATION/ENSEIGNEMENT (premier fascicule)
41	RESEAUX
42	SUITES
43	CINEMATIQUE RELATIVISTE
45	PETITES variations OU L'ART DE DERIVER SANS LE SAVOIR
46	NOMBRES A L'ECOLE ELEMENTAIRE (seconde édition corrigée)
47	MATERIAUX POUR LOGO
48	MESURE DES LONGUEURS ET DES AIRES
50	GROUPE AVIATION ET ENSEIGNEMENT (second fascicule)
51	INSTRUMENTS DE GEOMETRIE

Reproduction de textes anciens

- I- DISME (Simon STEVIN)
- II- GEOMETRIE ELEMENTAIRE (F. KLEIN) - 1846 -
- III- DICTIONNAIRE DE MATHEMATIQUE (J. OZANAM) - 1691 - (Tome 1)
- IV- " " " " " " " " " " " " " " (Tome 2)

EDITEUR : I.R.E.M de Paris-Sud

DIRECTEUR/RESPONSABLE DE LA PUBLICATION : François COLMEZ

Tirage : 200 exemplaires

Dépôt légal : 2 88612 019 1