UNIVERSITÉ PARIS XI

U.E.R. MATHÉMATIQUE

91405 ORSAY FRANCE

N° 121-75.27

LIMA ▽

PAR

PATRICK MERISSERT-COFFINIERES

LIMA ▽

Patrick MERISSERT-COFFINIERES *

=-=-=-=-=

SOMMAIRE :

* C.N.R.S.
Laboratoire AL KHOVARIZMI

I- MOTIVATIONS.-

LIMA Δ est l'une des dernières réalisations de la première phase du projet LIMA. On peut même le considérer comme un pont vers la deuxième, puisqu'il s'agit en fait d'un langage dans lequel l'utilisateur est censé écrire des programmes simples qui seront, finalement, de petits "sous-LIMA".

Initialement, il s'agissait simplement d'écrire un système reliant les différents constructivismes en usage, les manipulant, et les traduisant les uns dans les autres :
Fonctions récursives
λ-calcul
Calcul combinatoire
Algorithmes de MARKOV , systèmes de POST
Machines de TURING, automates
Grammaires formelles

L'aspect hétérogène de ces sytèmes, et la recherche de points communs a amené à l'état actuel du projet : un langage aussi souple que possible, dans lequel l'utilisateur peut définir des systèmes tels que ci-dessus avec un minimum de travail.

II- RAPPELS THEORIQUES.-

Les premiers systèmes constructivistes apparaissent vers 1922 (PROST). Le problème commence à se poser quand certains mathématiciens font la remarque que, par exemple, disposer d'une démonstration de l'existence d'un objet ne "fournit" pas cet objet. La conséquence qu'ils en tirent est qu'il faudrait peut-être n'admettre comme démonstrations d'existence due celles "construisant effectivement" l'objet désiré.

Un autre aspect du même problème est qu'un sous-ensemble A d'un ensemble E n'est utilisable que si l'on dispose de la description d'une méthode explicite pour déterminer si un élément de E est ou non dans A .

Une autre remarque fondamentale est que de telles restrictions ne sont pas génantes car un mathématicien ne fait jamais que manipuler un nombre fini de signes, ce qui est, en fin de compte, constructiviste.

A partir de ce moment, vont apparaître des sytèmes ayant pour but de formaliser les mathématiques à l'aide d'étapes purement constructivistes. Les plus célèbres sont les fonctions récursives, et les machines de Turing qui vers 1936, marquent, avec la thèse de Church, l'épanouissement du constructivisme mathématique.

- Dans tous les cas, il s'agit de fournir une formalisation de la notion intuitive de fonctions calculables.

Le premier essai est celui des fonctions récursives, commençant par la construction des fonctions primitives récursives : (il s'agit de fonctions de nombres entiers, à valeurs entières). Les fonctions de base dont les suivantes :

a) la fonction successeur :

$$N \to N$$
$$n \rightsquigarrow n+1$$

b) les fonctions constantes (les fonctions nulles suffisent)

$$N^p \to N$$
$$(n_1 - n_p) \rightsquigarrow a$$

c) les projections

$$N^p \to N$$
$$(n_1 - n_p) \rightsquigarrow n_k$$

Les fonctions primitives récursives seront les fonctions obtenues à partir de ces fonctions de base par composition, substitution, et par application du "schéma de récursion primitive" : Si h est primitive récursive à $k+1$ variables, et y primitive récursive à $k-1$ variables, alors la fonction f est primitive récursive.

$$f(O, x_2 \ldots\ldots x_k) = g(x_2, \ldots\ldots, x_k)$$
$$f(g+1, x_2 \ldots\ldots x_k) = h(y, f(y, x_2, \ldots\ldots x_k), x_2, \ldots\ldots x_k)$$

Nous disposons maintenant d'un ensemble dénombrable de textes. Nous remarquons déjà que ce sont, non pas des fonctions, mais des instructions de calcul pour des fonctions, et qu'une fonction donnée possèdera plusieurs textes permettant de la calculer, en fait une infinité (nous pouvons par abus de langage, et par analogie avec l'informatique parler d'algorithmes pour ces textes).

Nous pouvons numéroter ces algorithmes (par exemple en utilisant un ordre alphabétique) Soit f_i la fonction calculée par le i^e algorithme . Nous allons montrer qu'il existe des fonctions correspondant à la notion intuitive de fonction calculable (on peut donner des instructions permettant de les calculer) qui ne sont pas primitives récursives :

Définissons la fonction f de la manière suivante :

$$f(n) = f_n(n)+1$$

Alors f ne peut être égale à aucun f_p car

$$f_p(p) = f_p(p)+1$$

est impossible.

Remarques : L'argument de "diagonalisation" utilisé içi est un outil fréquemment utilisé dans les problèmes de récursivité.

Nous avons dû introduire une numérotation pour démontrer un résultat qui en est indépendant. Cette technique sera également rencontrée fréquemment.

La notion de récursive primitive est donc insuffisante pour traduire la notion intuitive de fonction calculable. Mais il semble que l'argument diagonal utilisé ci-dessus puisse être utilisé pour toute autre notion analogue. La conséquence à en tirer est que la classe des fonctions calculables et partout définies ne peut être énumérée. Pour tourner la difficulté il faut admettre des fonctions "partielles", c'est-à-dire dont le domaine n'est pas N tout entier.

Elles seront introduites par le moyen suivant :

On ajoute aux schémas de formation des primitives récursives l'opérateur μ dit "opérateur du moindre nombre".

Si $Q(z,x_1,\ldots x_n)$ est un prédicat de n+1 variables, alors $\mu_z(Q(z,x_1,\ldots x_n))$ est une fonction à n variables x_1, x_2, x_n dont voici une méthode de calcul :

$x_1 \ldots x_n$ étant fixés , on considèrera la proposition $Q(0,x_1,\ldots x_n)$. Si elle est vraie $\mu_z(Q(z,x_1,\ldots x_n)) = 0$. Sinon on considère $Q(1,x_1,\ldots x_n)$ etc..... .

$\mu_z(Q(z,x_1 \ldots x_n))$ sera égal au moindre z tel que $Q(z,x_1,\ldots x_n)$ soit vrai, s'il en existe un(d'où son nom) et non définie s'il n'en existe pas.

Si f est une fonction de n+1 variables, nous poserons

$$\mu_z(f(z,x_1 \ldots x_n)) = \mu_z((f(z;x_1 \ldots x_n)=0))$$

(l'on convient que si $f(z,\ldots)$ n'est pas défini $(f(z,\ldots)=a)$ est faux pour tout a) .

Alors les fonctions "générales récursives" (ou simplement "récursives") seront les fonctions obtenues à partir des fonctions de base (a) b) c)) par composition, substitution, le schéma de récursion primitive et le schéma suivant :

Si φ est une fonction récursive à N+1 variables, alors

$$f(x_1,\ldots x_n) = \mu_z(\varphi(z,x_1 \ldots . x_n))$$

est une fonction récursive.

Nous allons maintenant donner une description rapide de l'autre système constructiviste important : les machines de Turing.

Une machine de Turing est, fondamentalement une "boite noire" à travers laquelle passe un (ou plusieurs) ruban, suite de cases sur lesquelles peuvent être inscrits des symboles. Une case seulement à la fois est accessible à la machine. Il y a un nombre fini de symboles possibles, et la machine peut prendre un nombre fini d'états internes (dont un est en général dit "initial" et certains dits "finaux", terminologie qui s'explique d'elle même).

La machine opère de la façon suivante :

Elle "lit" le symbole ou l'absence de symboles sur la case accessible, a alors le choix entre :

écrire un nouveau symbole sur cette case
effacer le symbole écrit
ne rien faire

Puis a le choix entre

déplacer d'un cran en avant le ruban
le déplacer d'un cran en arrière
ne rien faire.

Et enfin, passer à un nouvel état (fonction de l'état précédent et du symbole lu).

Au démarrage, la machine est dans l'état initial, le "calcul" est considéré fini quand elle atteint un état final. Bien sûr diverses variantes sont possibles : on peut apporter des restrictions aux possibilités ci-dessus (interdiction de reculer, ou d'effacer, etc...). Ou, au contraire, on peut introduire un élément aléatoire, la machine à un moment ayant plusieurs décisions possibles (machines non déterministes).

Il faut alors fixer une convention décrivant comment une telle machine calcule une fonction. Voici l'une des plus simples :

L'alphabet des symbole sera réduit à {1}.

Au départ le ruban porte n"1" à la suite, le "lecteur" de la machine étant à droite du dernier 1 . Quand le "calcul" est fini, le ruban porte f(n)"1". Si le calcul ne se finit jamais, f(n) est indéfinie .

Pour définir des fonctions à plusieurs variables avec un seul ruban (ce qui est le cas dans les machines de Turing normales) il suffit d'ajouter un symbole $*$ à l'alphabet et $n_1, n_2, \ldots, n_p$ sera représenté sur le ruban par $\underbrace{11\text{-}1}_{n_1} * \underbrace{11\text{-}1}_{n_2} * \underbrace{11\text{-}1}_{n_p}$

Le résultat fondamental du constructivisme est que ces deux tentatives de définition de fonctions calculables (récursives, et Turing-calculables) ainsi que toutes les autres, aboutissent aux mêmes résultats.

Voici enfin une description de deux autres formalismes :

λ-Calcul :

L'alphabet est le suivant : un certain nombre de lettres appelées variables (XYZ...), les parenthèses et λ . L'opération de base est la concatenation : (FG)
Ex : (XY) ; ((XY)Z) ; ((Y((XX)(ZY)))(TX)).

Une convention de simplification est que :
(XYZ) est lu comme ((XY)Z),
ce qui peut se traduire en disant que l'on "associe vers la gauche".

On obtient ainsi toutes les formules sans "λ" .

Une formule élémentaire avec "λ" est de la forme (XλF) . X étant une variable et F une formule quelconque ; (l'écriture courante dans la littérature est (λX|F) mais nous adoptons içi une écriture par avance adaptable à la syntaxe APL !)

La définition récursive d'une formule correcte est :

$$F = X \mid (FG) \mid (X\lambda F)$$

Définissons maintenant une relation d'équivalence $\mathcal{R}$.
Si $\Sigma\,[G,X,F]$ est le résultat de la sustitution de la formule G à la variable X dans F

α) $(X\lambda F)\,\mathcal{R}\,(Y\lambda\Sigma[Y,X,F])$ si Y n'apparait dans F que comme variable muette.

β) $((X\lambda F)G)\,\mathcal{R}\,\Sigma\,[G,X,F]$

(Dans (XλF) la variable X est dite muette).

Bien sur, la relation $\mathcal{R}$ doit être compatible avec la concatenation, ce qui achève de la définir.

Comment définir des fonctions calculables avec ce formalisme ? . Les formules

$$(XY\lambda\,(X(X(X....(XY)))))$$

n-fois

sont considérées représenter l'entier n , et notées n^* . Alors soit F une formule ; si pour tout n^* (Fn^*) est équivalent à un p^* , F représente une fonction calculable.

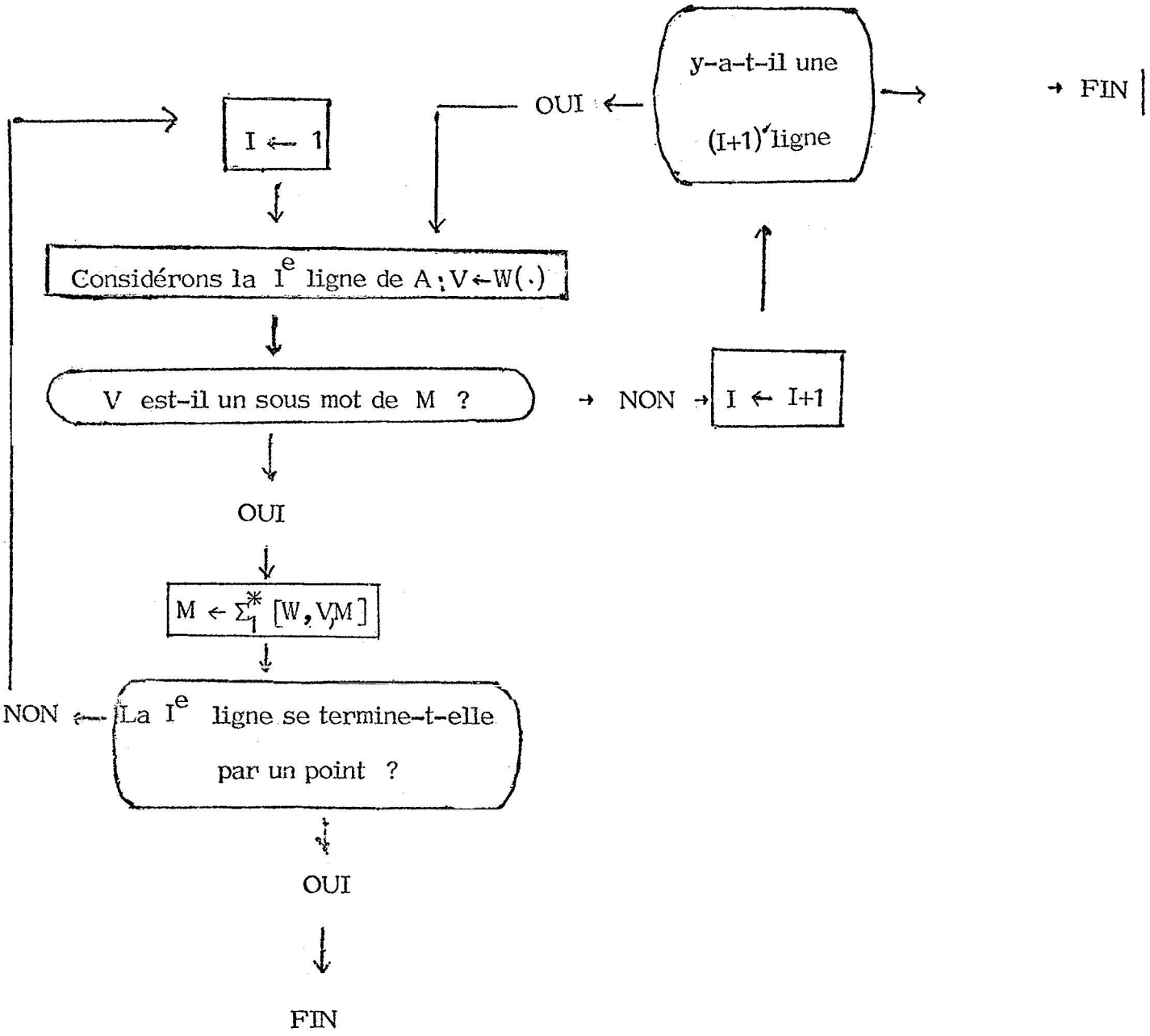
ALGORITHMES DE MARKOV.-

Un algorithme de Markov sur un alphabet A est une suite de lignes de la forme :

$$V \to W \quad \text{ou} \quad V \to W \ .$$

V et W étant des mots de A (éventuellement vides) .

L'application d'un algorithme à un mot M peut être décrite par l'organigramme suivant :



On peut considérer que seule la "FIN" découlant d'une ligne pointée est une fin légale. Il est aisé de représenter des fonctions par ces algorithmes.

$^{*}\Sigma_1$ n'opère les substitutions que sur la 1 ère occurence de V dans M .

Exemple : sur l'alphabet {1,* }

1 $\quad * | \to || *$

2 $\quad * \to .$

3 $\quad | \to * |$

Si le mot $\underbrace{||| \; ||}_{\text{n fois}}$ représente le nombre n , alors on voit aisément que l'algorithme représente la fonction f telle que f(n)=2n .

Exemple d'application :

|||

3 $\quad * |||$

1 $\quad ||*||$

1 $\quad |||*|$

1 $\quad ||||*$

2 $\quad |||| .$

3 METHODES

LIMA▽ est principalement un système manipulant des vecteurs de caractères (dans une version perfectionnée, les techniques d'APL GA pour les tableaux de tableaux pourraient être introduites).

La clef du système est l'introduction d'un nouveau type de variable LIMA▽ :

Il n'est pas surprenant que ce type ait été inspiré par l'un des constructivismes classiques, les systèmes de POST, qui sont, rappelons le, une variation des algorithmes de MARKOV.

Le format de ces variables est le suivant :

Une liste non vide de vecteurs de caractères munie d'un pointeur

```
xxxxxxx
xxx          p ≥ O
xxxx
xxxxx
xx           n ≥ O
xxxx
```

Une telle variable peut être engendrée soit par un mode spécial d'input, soit à partir de ses vecteurs composants par des opérateurs de composition.

Ce n'est pour l'instant qu'un assemblage, faiblement structuré, de caractères. La ou les "interprétations naturelles" viendront des opérateurs primitifs qui accepteront ces variables comme arguments.

Les plus simples et les moins intéressants, quoique nécessaires, sont des opérateurs de décomposition en vecteurs. (Par eux, l'utilisateur peut définir des fonctions créant des interprétations personnelles).

Les plus intéressants donnent l'interprétation annoncée, (liée aux systèmes de POST). Avant de les décrire, il faut mentionner qu'ils ne s'appliquent que dasn un "environnement", c'est à dire en fonction d'une variable-système ⎕AL (vecteur de caractères) qui représente l'alphabet avec lequel sont composés les mots manipulés. Tou caractère extérieur à cet alphabet sera considéré comme une lettre "symbolique".

Voyons maintenant l'interprétation. Les lignes jusqu'au pointeur sont simplement des vecteurs de caractère, écrits avec l'alphabet et les "symboles". La ligne pointée est le "résultat".

Les lignes ultérieures sont des "prédicats" LIMA ▽ sur les symboles intervenant plus haut, qui représentent des assemblages de l'alphabet. Voyons en détail ce que cela signifie, en distinguant deux cas :

1) le pointeur désigne la 1' ligne

Dans ce cas, la variable "représente" une fonction qui, appliquée à un vecteur de caractères dira si celui-ci est de la forme indiquée par la 1ère ligne après que les symboles aient été remplacés par des assemblages vérifiant les pré-

dicats des lignes ultérieures.

Exemple : Si ⎕AL←'X Y Z ='

```
→ | α = α
  | 1 = ρ Δ α
  |'='  ≠ Δ α
```

Les vecteurs 'X=X' 'Y=Y' 'Z=Z' seront les seuls acceptés. (Δ suivi d'un symbole est un type de nom qui n'est pas utilisé normalement en LIMA ∇ . Pour le reste les lignes après le pointeur sont des lignes LIMA ∇ normales).

2) Le pointeur désigne une ligne p+1 (p>0)

Cette fois-çi la variable "représente" une fonction p-adique : Si les p vecteurs proposés sont admissibles (au sens 1^e ci-dessus) comme étant les p premières lignes, on engendre la p+1^ème ligne avec les mêmes valeurs des symboles

Exemple : Si ⎕AL ← 'XYZ → '

```
  | 1 → 2
  | 2 → 3
→ | 1 → 3
  | (1 = ρΔ1) ∧ (1=ρΔ2)∧(1=ρΔ3)
  | ('→' ≠ Δ 1) ∧('→' ≠ Δ2)∧('→' ≠ Δ3)
```

Alors, à 'X→Y' et 'Y→Z'correspondra 'X→ Z'

La p+1^e ligne peut contenir des symboles n'apparaissant pas dans les p premières, si parmi les lignes ultérieures, il en est une de la forme

Δ α ← [expression fonction des autres symboles]

α étant le symbole en question.

Les détails sur l'exécution de ces opérateurs, diagnostics d'erreur, etc..... seront donnés dans la paragraphe sur la sémantique (IV.4).

Nous nous contenrerons içi de remarquer que ce formalisme nous fournit un outil à la fois puissant et versatile pour le but recherché : nous verrons dans le chapitre VI comment il permet d'engendrer rapidement aussi bien un analyseur syntactique qu'un programme admettant et effectuant des algorithmes, etc.....

4.- LINGUISTIQUE.-

1) Alphabet

LIMA vutilise les lettres, soulignées ou non, Δ et ⍙ (qui ne seront pas considérées comme des lettres), les chiffres, et les symboles suivants :

¨¯<≤=≥>≠∨∧-+÷×∈ρ~↑↓⍳○⊖→←⌈⌊∇∘'⍎⍕(⌈)⌋⊤|,/\⌽⊖

2) Lexique

Les noms de variables ou de fonctions sont les mêmes qu'en APL (grâce à l'opérateur ⍎ !)

3) La syntaxe est, fondamentalement la syntaxe APL : Il v a comme en APL 3 sortes d'êtres :

variables-fonctions (primitives ou non)-séparateurs

Les séparateurs (() ,[],') fonctionnent exactement comme en APL

Comme en APL, les opérateurs primitifs peuvent être monadiques, dyadiques, ou les deux à la fois.

Comme en APL, les fonctions non primitives peuvent être monadiques ou dyadiques, ou niladiques (ou exclusif)

Contrairement à APL, le résultat d'une fonction (primitive ou non) ne peut jamais être une fonction (dans APL / \ . et ∘ : étaient des fonctionnelles)

4) sémantique

Les variables ont des valeurs de 3 types

1: entiers relatifs

2: caractères

3: le type spécial défini au chapitre 3

Ces variables peuvent être en scalaires ou en vecteurs

→← et ⍬ ont la même signification.

Δ est un opérateur monadique, exigeant des entiers, qui déclenche simultanément un mode d'output :

Il sert à rentrer des variables de type 3 de la façon suivante :

```
      Λ←Δ1  2
[1]       α=α
[2]       α→α
          0≠ρΔα
          Δ
[1]       α
[2]       α→ω
[3]       ω
          Δ
```

Si l'utilisateur a demandé une variable à n antécédents n>O , il peut diminuer ce nombre en ne rentrant rien en face de [p], pour p≤n ; mais la ligne [n+1] doit recevoir un vecteur de caractères non vide, sous peine d'un diagnostic d'erreur :

```
      Λ←Δ2
[1]       X
[2]
[3]
THIS LINE CANNOT BE EMPTY
[3]       XX
          Δ
```

Les opérateurs +X - ¯|≤<≥>∨∧~⍳ ⌽ sont les mêmes qu'en APL.

Les opérateurs = ≠ ρ↑↓ ∈, ⌿ / sont les mêmes qu'en APL, mais étendus aux variables de type 3

L'opérateur ÷ représente la division arithmétique

L'opérateur ⍣ est un opérateur monadique ayant pour résultat une fonction monadique. Il exige comme argument un vecteur de deux caractères distincts. La fonction résultat analysera un vecteur de caractères en considérant le couple de caractères initiaux comme des séparateurs :

Exemples

```
      F←⍣'⊂⊃'
      F'1⊂⊂ΛC⊃+0⊃⍳0'
0 0 1 2 2 2 1 1 1 0 0
      F'Λ+⊂1×1⊃⊃'
1 1 1 2 2 2 2 1
```

L'opérateur \ est dyadique. Il admet comme argument de droite un vecteur de caractères et comme argument de gauche un entier 0 1 ou 2 . Il supprime les blancs de la manière suivante :

1 supprime les blancs initiaux

2 supprime les blancs finaux

0 supprime les blancs ne séparant pas des "vrais" caractères

L'opérateur ⍴

a) monadique, il transforme un vecteur en une variable de type 3 ayant le vecteur comme seule ligne.

b) dyadique : il "imbrique" vecteurs et variables de type 3 pour former des variables plus grandes :

α) Si les deux arguments sont des vecteurs, il forme la variable dont le premier vecteur est le seul antécédent et le deuxième le résultat.

β) Si le premier argument est une variable de type 3 et le deuxième un vecteur, il forme la variable ayant le vecteur comme antécédent supplémentaire.

γ) Si les arguments sont inversés par rapport à β) il forme la variable ayant pour prédicat supplémentaire le vecteur.

L'opérateur ⍴ est en quelque sorte l'inverse du précédent.

a) monadique : appliqué à une variable de type 3 sans antécédent il sort le lecteur résultat.

b) dyadique : il admet comme arguments un entier et une variable de type 3

α) le premier argument est entier : n

si n = 0 le résultat est le nombre "d'antécédents" de la variable

si n > 0 le résultat est le n^{e} "antécédent" ou 0

si n < 0 le résultat est la variable moins le n^{e} "antécédent" (ou la variable elle même s'il n'existe pas).

β) le deuxième argument est entier n

si n = 0 le résultat est le nombre de "prédicats"

si n > 0 le résultat est le n^{e} "prédicat" ou 0

si n < 0 le résultat est la variable moins le n^{e} "prédicat" (ou la variable elle même s'il n'existe pas) .

⌈ est un opérateur de substitution .

Il est dyadique, ses arguments sont deux vecteurs de caractères, son résultat une fonction monadique

Exemple :

```
      F←'AB'⌈'BC'
      F'IIBCBC'
IIABAB
```

•• est un opérateur d' abstraction fonctionelle

Il fonctionne dans les mêmes conditions que le précédent, à la différence que l'argument de droite doit être composé d'un seul caractère .

Exemple :

```
      G←'X'••'1+3×X'
      G 0
1
      G 1
4
```

On voit que c'est un opérateur d'une puissance sémantique comparable à ⌽ .

⌊ est un autre opérateur de substitution, dyadique .

Son argument de droite est un vecteur de caractères

Son argument de gauche est un vecteur de variables de type 3 sans antécédent (la présence ou l'absence de prédicats ne jouera aucun rôle, l'opérateur ⌊ interprétant le vecteur de variables comme un vecteur de vecteurs)

Il s'exécute en fonction de ce que l'on a appelé l'environnement.

Les "symboles" dans le deuxième argument (c'est à dire les caractères qui ne sont pas dans ⎕AL) sont remplacés dans l'ordre par les vecteurs de premier argument. S'il y a moins de vecteurs que de symboles, les derniers symboles sont laissés inchangés.

Exemples :

```
      ⎕AL
XYZSP
      A←A3⍴0
[1]       !!
          Λ
[1]       ??
          Λ
[1]       __
          Λ
      A⌊'XαYωαXX1'
X!!Y??!!XX__
```

o est l'opérateur fondamental d'interprétation des variables de type 3 .

a) monadique il s'applique à un vecteur de variables de type 3 au nombre d'antécédent constant égal à O 1 ou 2 ; son résultat est une fonction monadique dans les deux premiers cas, dyadique dans le troisième. Le résultat de cette fonction a été décrit dans le chapitre 3 .
Au moment de l'exécution de cette fonction résultat, la liste des "prédicats" d'une variable est considérée comme un programme LIMA∇ et exécutée ligne par ligne.
Si une ligne ne peut pas être exécutée, on obtient un diagnostic de "DOMAIN ERROR" sur la fonction.

b) dyadique, son premier argument est une variable de type 3 (ou un vecteur de telles variables de nombre d'antécédents constant) à n antécédent (n > O) .
Le deuxième argument est un vecteur de caractères.
Le résultat est une variable de type 3 à n-1 ou n antécédents obtenue de la façon suivante :
On essaie de considérer le vecteur comme "admissible" pour l'antécédent de la variable. En cas d'échec, le résultat est la variable elle même. En cas de succés, les "valeurs" des symboles qu'on en déduit sont substituées dans les autres antécédents (et le "résultat"), et les prédicats ne contenant que ces symboles sont éliminés, ainsi, bien sûr, que la 1' ligne.

5.- PRAGMATIQUE

LIMA ∇ dispose d'un nombre restreint de commandes système, analogue à celles d'APL :

)CLEAR)VARS)FNS)OFF

An outre nous avons déjà constaté l'existence d'une variable système : ⎕AL dont la signification a déjà été exposée. Signalons que l'ordre :

⎕AL ← '.....'

n'est pas exécuté littéralement : en effet le caractère "blanc" doit obligatoirement appartenir à ⎕AL, et est automatiquement ajouté.

V.- TECHNIQUES ET ALGORITHMES.-

Une variable LIMAv est représentée internement par deux ou quelque fois trois descripteurs.

Le premier est un vecteur de quatre nombres décrivant la "forme" :

a b c d

a = 1 si c'est une fonction
O si c'est une variable

b : si c'est une variable b est égal à son type : 1 pour entiers
2 pour caractères
3 pour "variables type 3"

si c'est une fonction , b est égal à l'adicité (O,1 ou 2)

c : si c'est une variable, c vaut O ou 1 suivant que c'est un scalaire ou un vecteur.
Si c'est une fonction c vaut 1 ou O selon que le résultat est explicite ou non (terminologie APL)

d : si c'est une variable d est son ρ (le ρ d'un scalaire étant içi assimilé à O)
si c'est une fonction , d vaut 1 ou 2 suivant qu'elle est définie comme résultat d'un opérateur ou à l'aide du mode définition.

Le deuxième descripteur est très exactement le contenu de l'être LIMA v , ayant pour forme :

un vecteur pour les variables de type 1 ou 2
un tenseur de dimension 3 pour les variables de type 3
une matrice pour les fonctions.

Dans le cas d'une variable de type 3 un vecteur supplémentaire fournit l'information relative aux "pointeurs".

On voit aisément que cette description est redondante. Cette redondance est peu coûteuse et facilite grandement la programmation.

ORGANISATION GENERALES DU SYSTEME LIMA ▽

Le premier programme qui prend la main est ANALYSE

En présence d'une ligne, il la classe dans l'une des quatre catégories suivantes :

1) mode définition
2) commentaire
3) commande système
4) ligne de mode exécution ordinaire

Dans le cas 1 ANALYSE traite la ligne directement

Dans le cas 2 il n'y a rien à faire

Dans le cas 3 ANALYSE donne la main à EXCOM

Dans le cas 4 ANALYSE donne la main à ANAL

Le premier programme utilisé pour ANA est INDIT: il décompose une expression en"élements" , un élément étant :

un identificateur

un vecteur ou scalaire explicite

un opérateur ou Δ ou ←

une expression entre parenthèses

une expression indexée.

Exemples : A+1 2×(V1-V2)[3]
1233345555555555

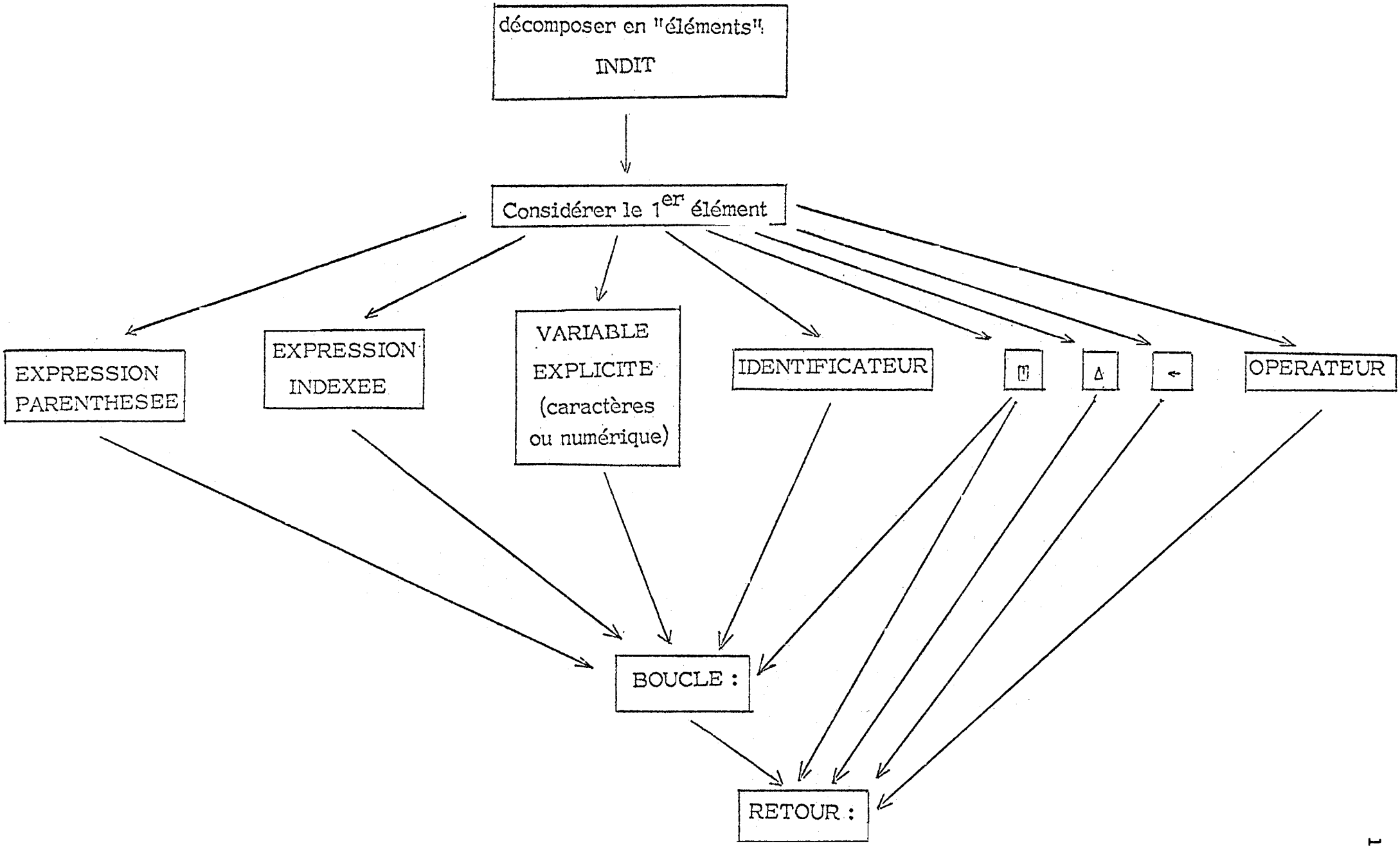
Si INDIT repère une erreur (il peut diagnostiquer certaines erreurs simples, en particulier de séparateurs) des "O" apparaîssent à partir de l'apparition de l'erreur

2 3) + A - B
OOOO 1 2 3 4

(comme toujours l'analyse se fait de la droite vers la gauche)

Les organigrammes suivants montrent l'organisation d'ANAL après INDIT :

Les tests d'erreur ont été en général omis dans les organigrammes.


décomposer en "éléments":
INDIT
Considérer le 1er élément
EXPRESSION PARENTHESEE
EXPRESSION INDEXEE
VARIABLE EXPLICITE (caractères ou numérique)
IDENTIFICATEUR
OPERATEUR
BOUCLE :
RETOUR :

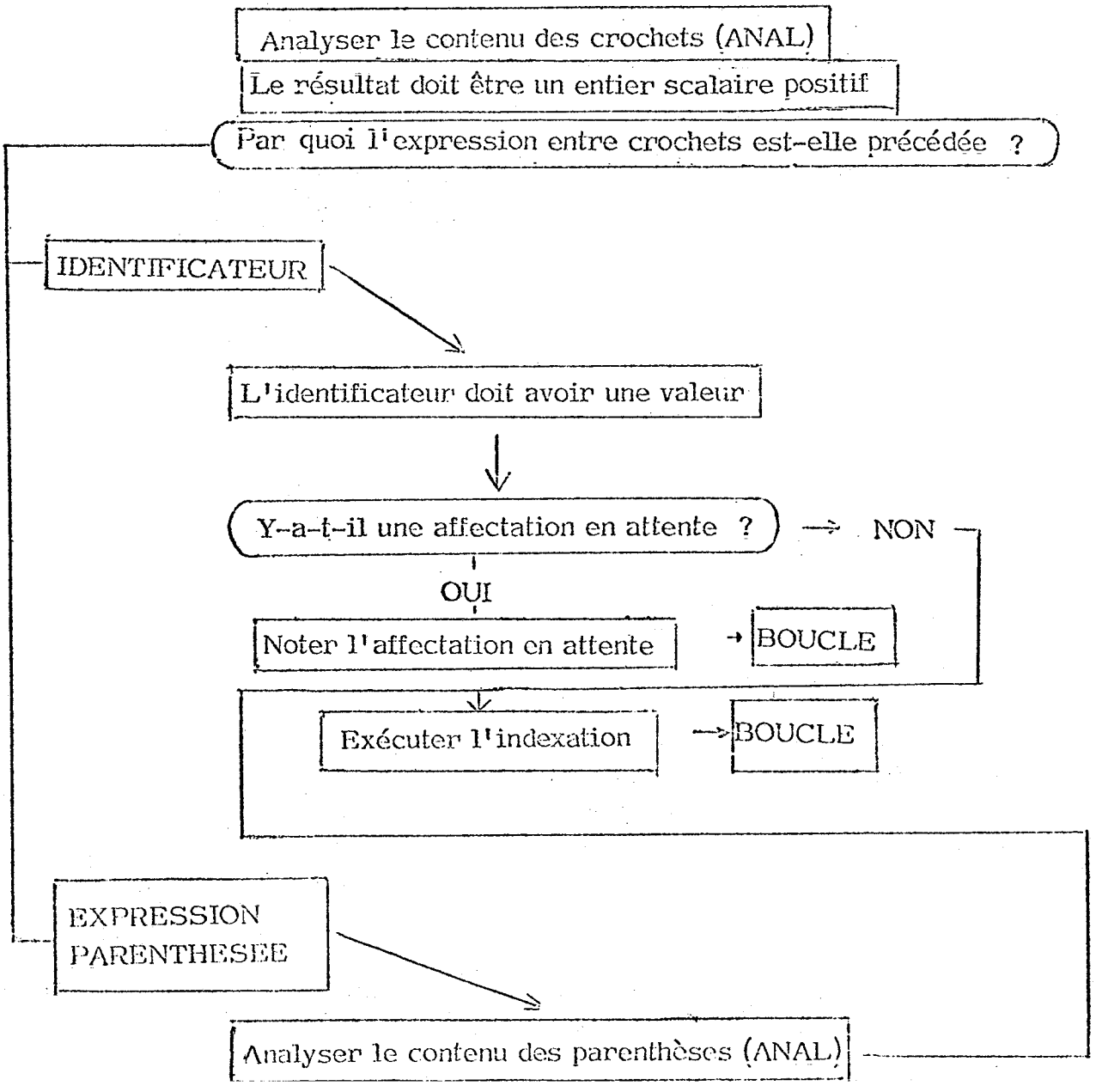
EXPRESSION PARENTHESEE

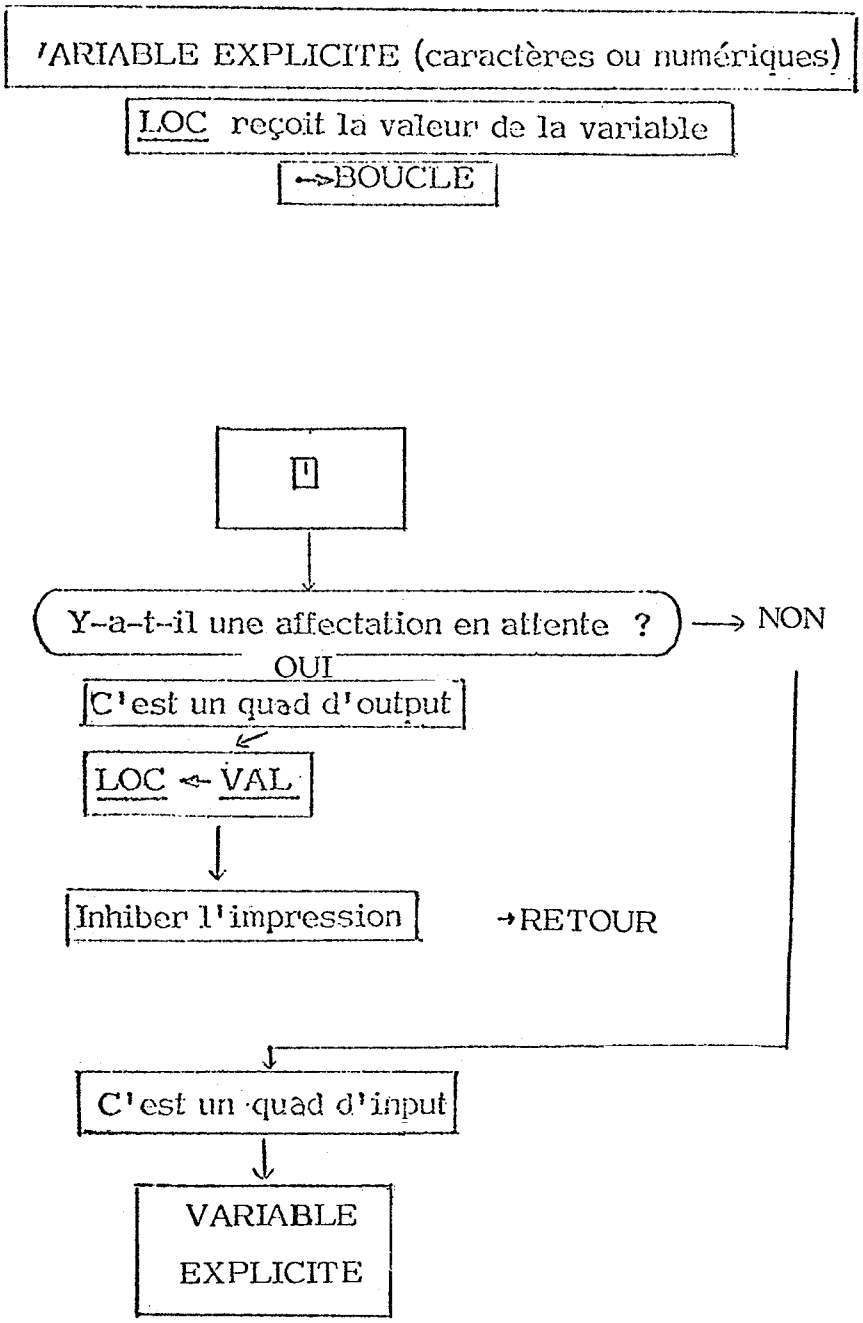
Analyser le contenu des parenthèses (ANAL)

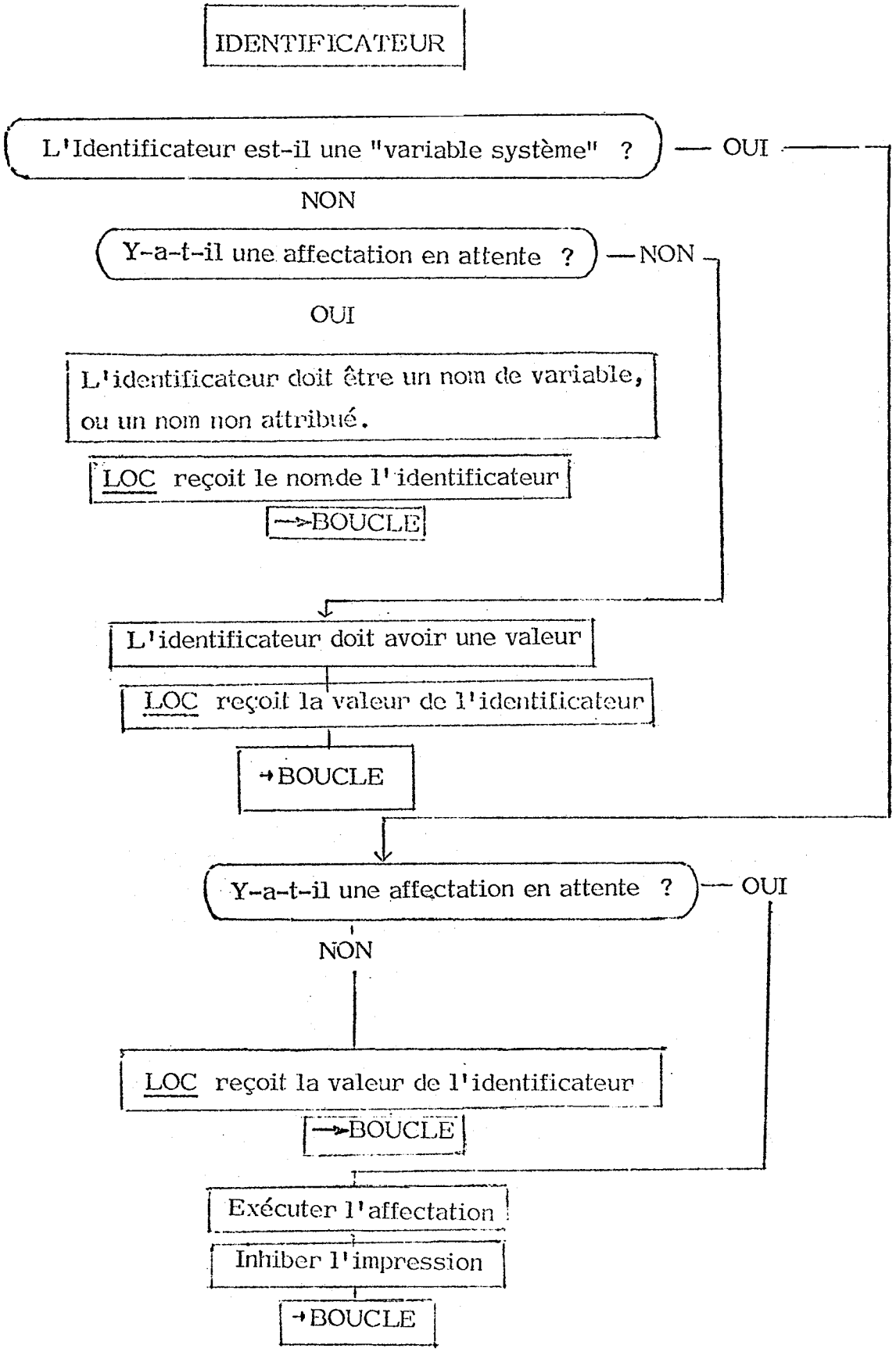
LOC reçoit le résultat

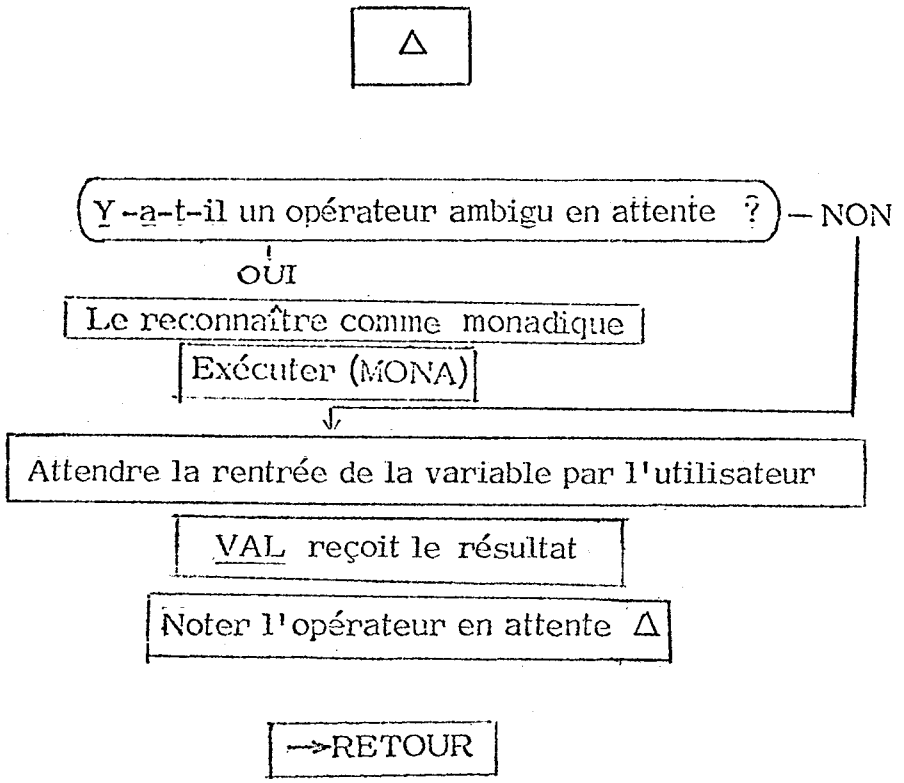
→BOUCLE

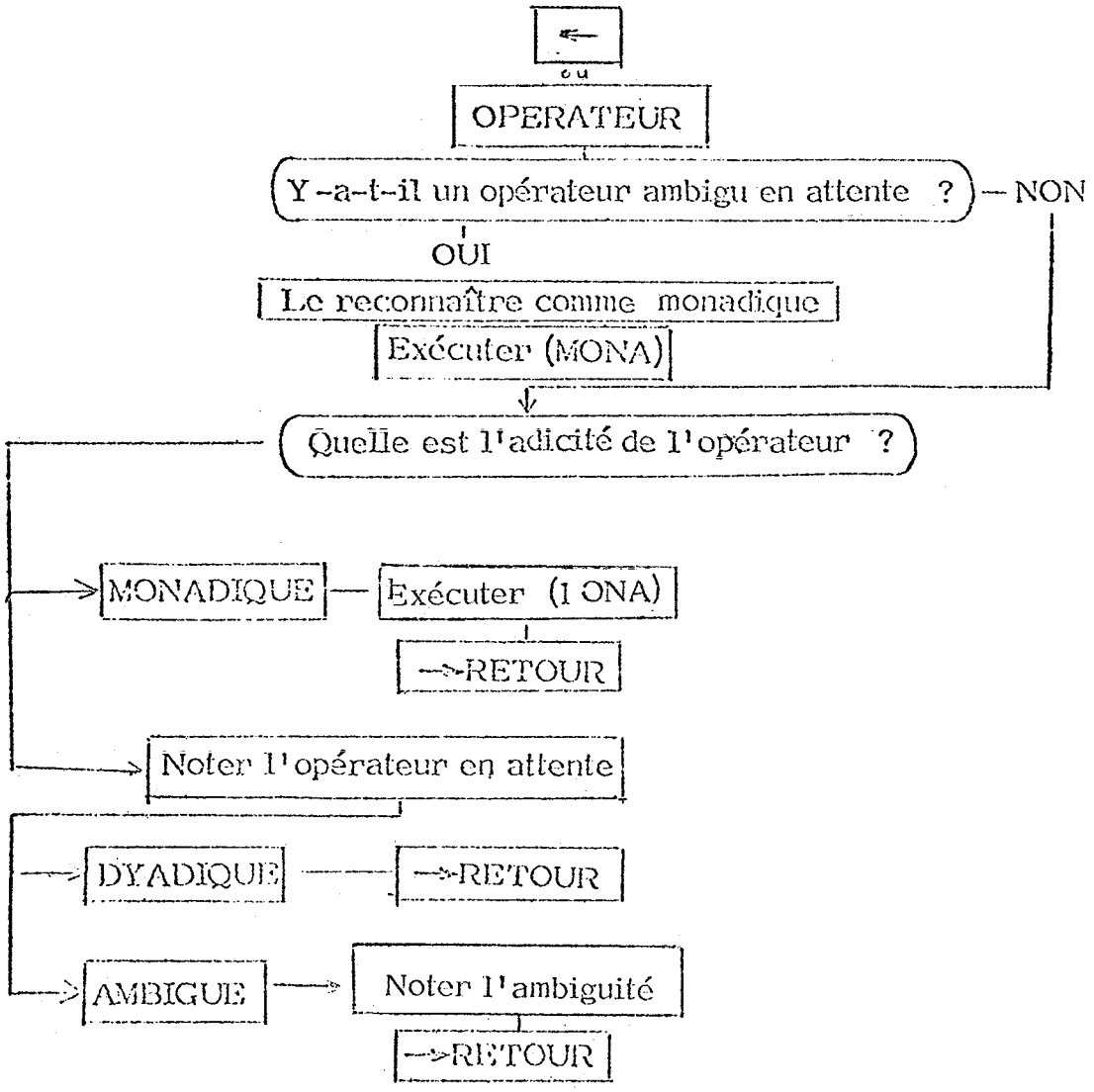
EXPRESSION INDEXEE

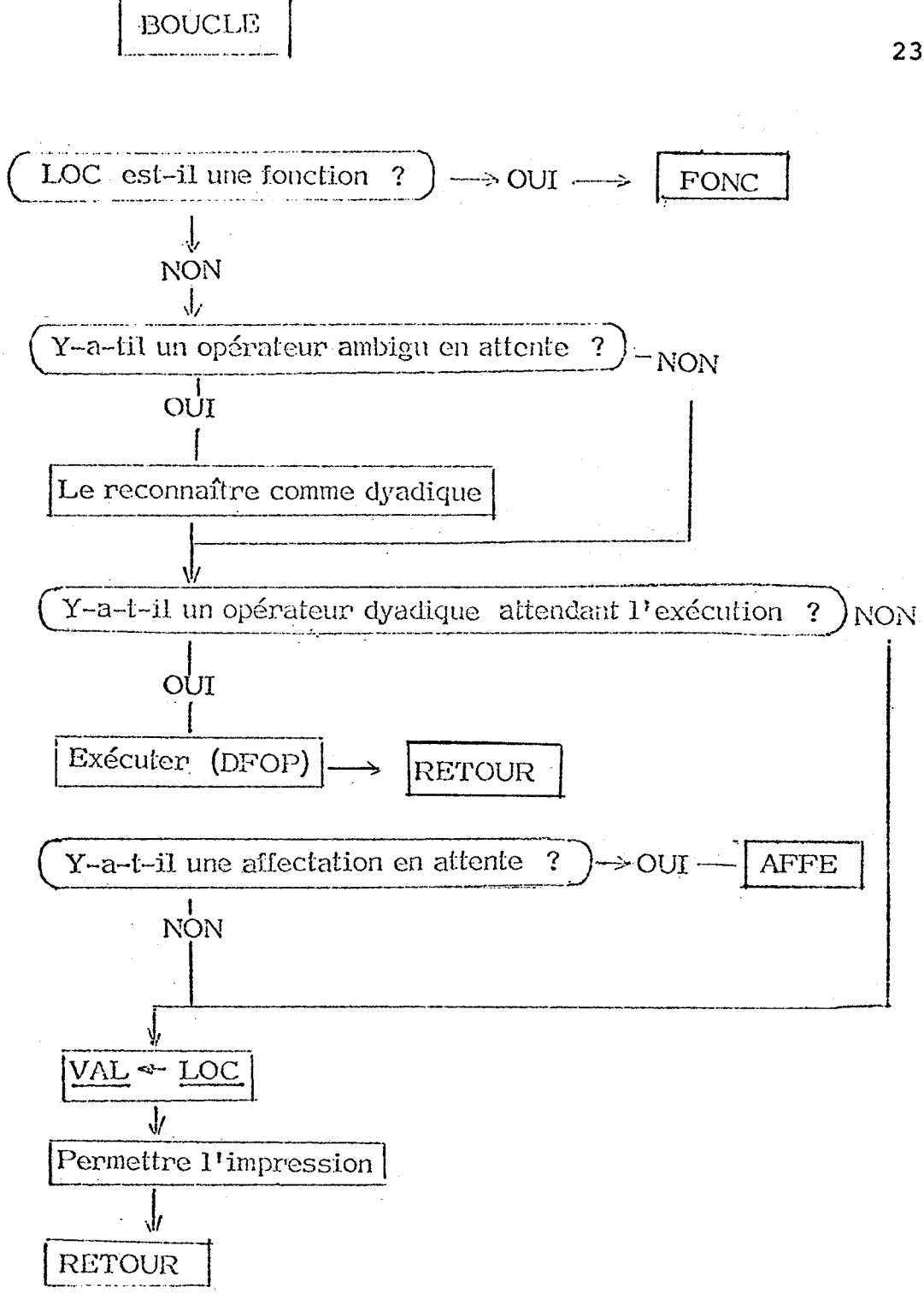


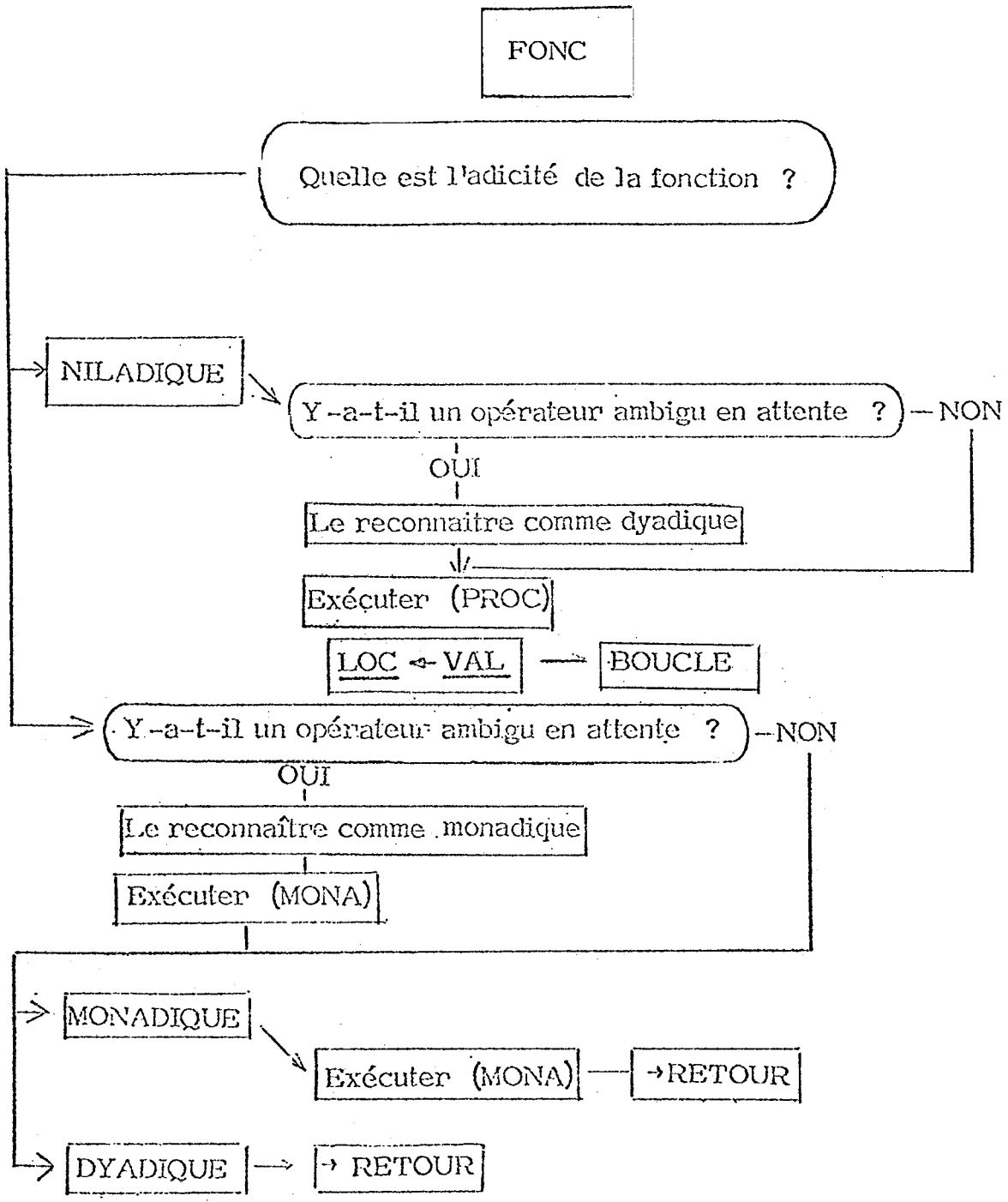

'ARIABLE EXPLICITE (caractères ou numériques)
LOC reçoit la valeur de la variable
←→BOUCLE
Y-a-t-il une affectation en attente ?
NON
OUI
C'est un quad d'output
LOC ← VAL
Inhiber l'impression
→RETOUR
C'est un quad d'input
VARIABLE EXPLICITE

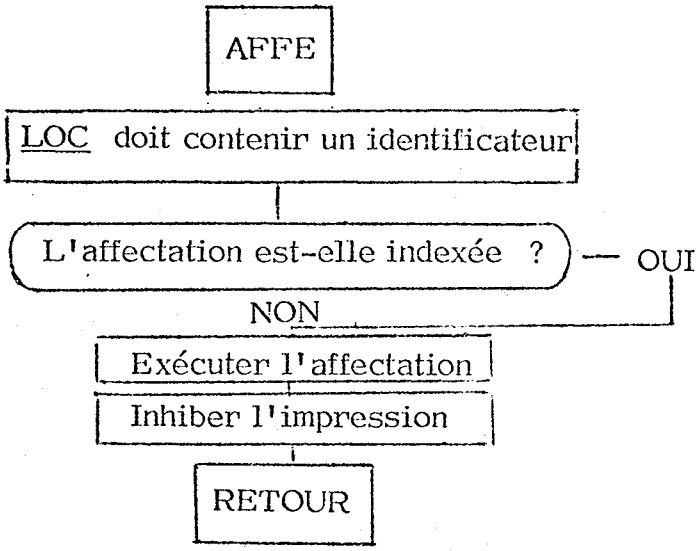

IDENTIFICATEUR
L'Identificateur est-il une "variable système" ?
OUI
NON
Y-a-t-il une affectation en attente ?
NON
OUI
L'identificateur doit être un nom de variable, ou un nom non attribué.
LOC reçoit le nomde l'identificateur
→BOUCLE
L'identificateur doit avoir une valeur
LOC reçoit la valeur de l'identificateur
→BOUCLE
Y-a-t-il une affectation en attente ?
OUI
NON
LOC reçoit la valeur de l'identificateur
→BOUCLE
Exécuter l'affectation
Inhiber l'impression
→BOUCLE

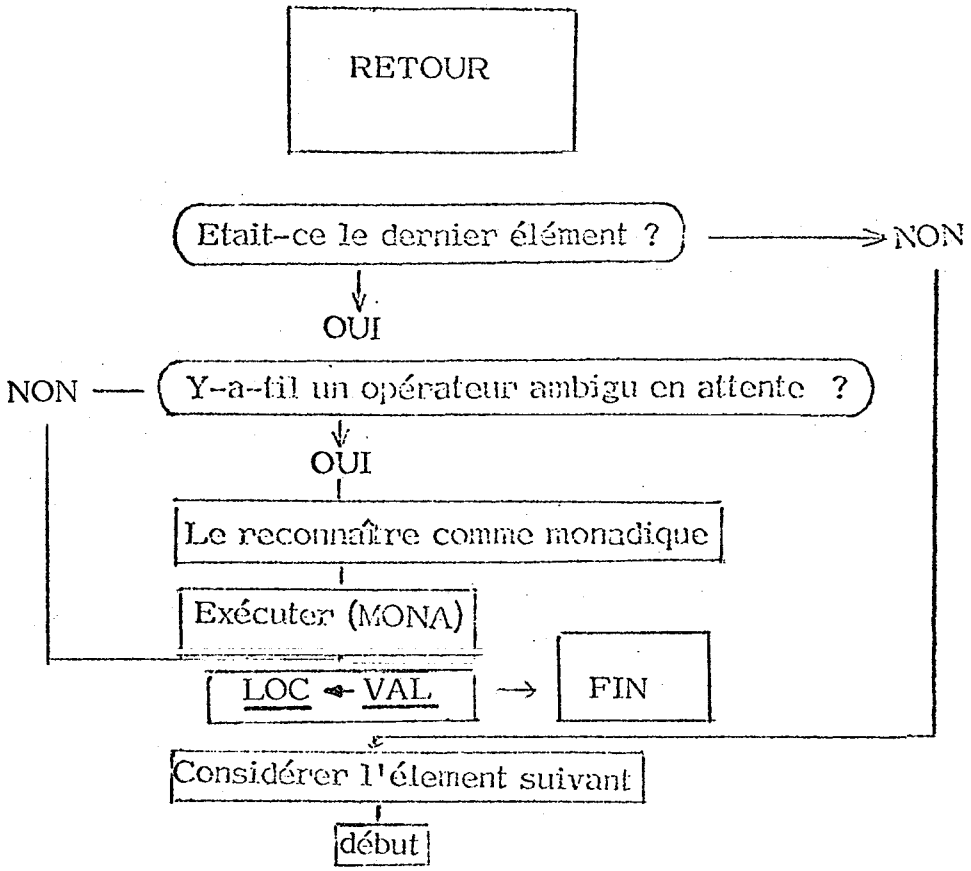

Δ
Y-a-t-il un opérateur ambigu en attente ?
NON
OUI
Le reconnaître comme monadique
Exécuter (MONA)
Attendre la rentrée de la variable par l'utilisateur
VAL reçoit le résultat
Noter l'opérateur en attente Δ
→RETOUR


←
ou
OPERATEUR
Y-a-t-il un opérateur ambigu en attente ?
NON
OUI
Le reconnaître comme monadique
Exécuter (MONA)
Quelle est l'adicité de l'opérateur ?
MONADIQUE
Exécuter (MONA)
→RETOUR
Noter l'opérateur en attente
DYADIQUE
→RETOUR
AMBIGUE
Noter l'ambiguité
→RETOUR


BOUCLE
LOC est-il une fonction ?
OUI
FONC
NON
Y-a-til un opérateur ambigu en attente ?
NON
OUI
Le reconnaître comme dyadique
Y-a-t-il un opérateur dyadique attendant l'exécution ?
NON
OUI
Exécuter (DFOP)
RETOUR
Y-a-t-il une affectation en attente ?
OUI
AFFE
NON
VAL ← LOC
Permettre l'impression
RETOUR


FONC
Quelle est l'adicité de la fonction ?
NILADIQUE
Y-a-t-il un opérateur ambigu en attente ?
NON
OUI
Le reconnaitre comme dyadique
Exécuter (PROC)
LOC ← VAL
BOUCLE
Y-a-t-il un opérateur ambigu en attente ?
NON
OUI
Le reconnaître comme monadique
Exécuter (MONA)
MONADIQUE
Exécuter (MONA)
→RETOUR
DYADIQUE
→ RETOUR


AFFE
LOC doit contenir un identificateur
L'affectation est-elle indexée ?
OUI
NON
Exécuter l'affectation
Inhiber l'impression
RETOUR


RETOUR
Etait-ce le dernier élément ?
NON
OUI
NON
Y-a-t-il un opérateur ambigu en attente ?
OUI
Le reconnaître comme monadique
Exécuter (MONA)
LOC ← VAL
FIN
Considérer l'élement suivant
début

DFOP

Identifier l'opérateur (primitif ou non)

Exécuter avec LOC et VAL comme argument

VAL reçoit le résultat

Permettre l'impression

MONA

Identifier l'opérateur (primitif ou non)

Exécuter avec VAL comme argument.

VAL recoit le résultat

Permettre l'impression

VI.- SESSION ET CONCLUSION.-

```
      TEST
     F←T'()'
     )FNS
F
     ⍝DEFINITION DU MODUS PONENS
     A←Δ 2
[1]     α
[2]     α→ω
[3]     ω
        (0≠ρΔα)∧0≠ρΔω
        0=(F Δα)[1]
        0=(F Δω)[1]
        Δ
     ⍝LES SEULES CONDITIONS SUR α ET ω SONT
     ⍝DES CONDITIONS DE SYNTAXE DE PARENTHESES(F)
     ⍝
     ∨
     ⍝D'AUTRES PREDICATS AURAIENT PU ETRE
     ⍝DEFINIS ET UTILISES.
     ⎕AL←'XYZT→()'
     MP←∘A
     (
     ∨
     '(X→X)'MP'(X←X)
     '(X→X)'MP'(X→X)→(XZ→XZ)'
(XZ→XZ)
     ⍝UTILISATION RECURSIVE DE L'OPERATEUR Δ
     ⍝POUR DEFINIR UN IDENTIFICATEUR DE MOTS
     ⍝DU LAMBDA-CALCUL

     A←Δ3⍴0
[1]     ⍵
        1=ρΔα
        Δα∈ALP
        Δ
[1]     (αω)
        ID Δα
        ID Δω
        Δ
[1]     (α∇ω)
        (1=ρΔα)
        Δα∈ALP
        ID Δω
        Δ
```

```
      ID←∘A
      ALP←'XYZSTUVWXYZSTUVW'
      ⎕AL←ALP,'()∇'
      )FNS
F ID MP
      ID'X'
1
      ID'((XZ)X)'
1
      ID'(Z∇(XZ))'
1
      ID'(X∇(XXX))'
0
      ⍝UTILISATION DE L'OPERATEUE ∧
      ⍝POUR DEFINIR UN SYSTEME ADMETTANT ET
      ⍝EXECUTANT DES ALGORITHMES DE MARKOV:


      ∇R←MARK
[1]     I←1,R←AR←⍳0
[2]     V←⍞
[3]     →(0=⍴V)/2
[4]     →(1≠⍴V)/7
[5]     →(V≠'↓')/ER
[6]     →FIN
[7]     →(~'→'∈V)/ER
[8]     →('.'≠V[⍴V])/S
[9]     AR←AR,⍕I
[10]     V←¯1↓V
[11]     S:J←V⍳'→'
[12]     R←R,('α',((J-1)↑V),'ω')∘.'α',(J↓V),'ω'
[13]     I←I+1
[14]     →2
[15]     FIN:R←R,∘.AR
[16]     →0
[17]     ER:'INCORRECT LINE'
[18]     →2∇


      ⍝CE PROGRAMME PERMET DE RENTRER DES ALGORITHMES
      ⍝DE MARKOV.VOICI LE PROGRAMME PERMETTANT DE LES
      ⍝EXECUTER:

   ∇R←A APPL V
[1]    AR←±⊖A[⍴A]
[2]    R←V
[3]    IF←¯1+⍴A
[4]    IN:I←1
[5]    R1←(∘.A[I])R
[6]    →(1≠⍴R1)/OUI
[7]    →(0=R1)/NON
[8]    OUI:R←R1
[9]    →(I∈AR)/0
[10]     →IN
[11]     NON:→(I=IF)/0
[12]     I←I+1
[13]     →IN+1
[14]     ∇
```

```
      ⍝EXEMPLE:ALGORITHME DE DOUBLAGE.


      R←MARK
*|→||*
|*→|.
|→*|
↓
      ⎕AL←'*|'
      R APPL '|||'
||||||
      ⍝ETANT DONNE LE PROGRAMME UTILISE,
      ⍝⎕AL NE DOIT CONTENIR AUCUN DES
      ⍝CARACTERES SUIVANTS: ⍺ ⍵ . ↓
      )OFF
```

La session ci-dessus donne une légère idée de la versabilité du système. Sur le plan de l'efficacité, les remarques suivantes s'imposent :
La possibilité de définir des fonctions non primitives, est dans ce système, d'intérêt fondamental. Dans la session que nous venons de voir, la plus grande partie du travail se passait à exécuter de telles fonctions, souvent récursives ! . Dans l' état actuel du système (interpréteur LIMA ∇) il y a donc presque en permanence une triple interprétation (à quoi s'ajoute, sur le plan encombrement la simulation de la récursivité APL)!. Aussi il n'est pas étonnant que la fabrication des analyseurs syntaxiques en particulier, donne des résultats très mauvais en efficacité. En vue d'obtenir un système vraiment opérationnel, une version compilateur de LIMA ∇ est en cours d'implémentation.

APPENDICE : QUELQUES LISTINGS

```
      ∇ANAL[⎕]∇
    ∇ ANAL V;CODE;VALΔ;VALIN;I;J;ROV;RCODE;W;VV;ADAP;M;M1;IS;II;NOΔ;NOIN;VALS
[1]     I←ROV←⍴V
[2]     CODE←¯1+VALΔ⍪VALIN←,0
[3]     →(I=0)/0
[4]     W←INDIT V
[5]     →(ERR=1)/SYN
[6]   INIT:→(V[I]='(∇[¯→:⎕')/OPAR,SYN,SYN,SYN,SYN,SYN,SYN
[7]     →(V[I]∈AL,CHI,OPRI,')]''Δ⍞')/5+⎕LC
[8]     MSERR←'SYMBOL ERROR'
[9]   LE:ERR←1
[10]    TERR←(2,ROV)⍴V,((I-1)⍴' '),'^',(ROV-I)⍴' '
[11]    →0
[12]    VV←(W=W[I])/V
[13]    →(V[I]=')]''Δ⍞')/PARF,CROF,QUO,DELT,QUAS
[14]    →(V[I]∈AL,CHI)/VAONO
[15]    →OPER
[16]  OPAR:MSERR←'PARENTHESIS ERROR'
[17]    →LE
[18]  SYN:MSERR←'SYNTAX ERROR'
[19]    →LE
[20]  PARF:→(W[I]=0)/OPAR
[21]    ANAL 1↓¯1↓VV
[22]    →(ERR=0)/6+⎕LC
[23]    J←W⍳W[I]
[24]    MA1←(2,J)⍴(J↑V),J⍴' '
[25]    MA2←(2,ROV+1-I)⍴((I-1)↓V),(ROV+1-I)⍴' '
[26]    TERR←MA1,TERR,MA2
[27]    →0
[28]    I←W⍳W[I]
[29]    →BOUCLE
[30]  QUO:→(VV[1]≠'''')/SYN
[31]    →(2|+/''''=VV)/SYN
[32]    VV←1↓¯1↓VV
[33]    VV←(0=-\''''=VV)/VV
[34]    LOCΔ← 0 2 ,(1≠⍴VV),⍴VV
[35]    LOCIN←VV
[36]    I←W⍳W[I]
[37]    →BOUCLE
[38]  DELT:→(CODE≠¯3)/6+⎕LC
[39]    CODE←RCODE
[40]    I←W⍳W[I]+1
[41]    MONA
[42]    →ERR/LE+1
[43]    I←W⍳W[I-1]
[44]    →(CODE≠0)/SYN
[45]    →(1=⍴VALΔ)/SYN
[46]    →(VALΔ[2]≠1)/DOER
[47]    →(∨/VALIN<0)/DOER
[48]    T← 0 0 0⍴' '
[49]    RVA←⍴VALIN
[50]    COM2←1
[51]  IF2:M← 0 0⍴' '
[52]    COM1←1
```

```
[53] IN1:⍞←RV←'[ ',(⍕COM1),'] '
[54]  VV←,(ρRV)↓⍞
[55]  →(0≠ρVV)/4+⎕LC
[56]  →(COM1≠VALIN[COM2]+1)/4+⎕LC
[57]  'THIS LINE CANNOT BE EMPTY'
[58]  →⎕LC-5
[59]  M←M LIS VV
[60]  →(COM1=VALIN[COM2]+1)/3+⎕LC
[61]  COM1←COM1+1
[62]  →IN1
[63]  VALIN[COM2]←(ρM)[1]-1
[64]  '⍞' '
[65]  VV←,8↓⍞
[66]  →(1≠ρVV)/2+⎕LC
[67]  →(VV='Δ')/FI1
[68]  M←M LIS VV
[69]  →⎕LC-5
[70] FI1:T←T PIL M
[71]  →(COM2=RVA)/FI2
[72]  COM2←COM2+1
[73]  →IN2
[74] FI2:VALΔ← 0 3 ,(1≠(ρT)[1]),(ρT)[1]
[75]  VALSD←VALIN
[76]  VALIN←T
[77]  →RETOUR
[78] VAONO:→(VV[1]≠' ')/2+⎕LC
[79]  VV←1↓VV
[80]  →(VV[1]='⎕')/VASY
[81]  →(M[I]=0)/SYN
[82]  →(' '∈VV)/NUM
[83]  →(VV[1]='¯')/NUM
[84]  →(VV[1]∈AL)/NOM
[85] NUM:LOCIN←,⍎VV
[86]  LOCΔ← 0 1 ,(1≠ρLOCIN),ρLOCIN
[87]  I←M⍳M[I]
[88]  →BOUCLE
[89] NOM:→(CODE≠21)/NOTAF
[90]  →(0=⎕NC 'Δ',VV)/2+⎕LC
[91]  →(1=⍎'Δ',VV,'[1]')/SYN
[92]  LOCΔ←,0
[93]  LOCIN←VV
[94]  I←M⍳M[I]
[95]  →BOUCLE
[96] NOTAF:→(VALER VV)/3+⎕LC
[97]  MSERR←'VALUE ERROR'
[98]  →LE
[99]  LOCΔ←,⍎'Δ',VV
[100] LOCIN←⍎'IN',VV
[101] →(3≠LOCΔ[2])/2+⎕LC
[102] LOCSD←⍎'SD',VV
[103] I←M⍳M[I]
[104] →BOUCLE
[105]FONC:→(LOCΔ[2]=0)/FNI
[106] →(CODE>0)/SYN
[107] →(CODE≠¯1)/3+⎕LC
[108]  CODE←0
[109]  →RETOUR
[110]  →(CODE≠¯3)/6+⎕LC
[111]  CODE←RCODE
[112]  I←M⍳M[I]+1
```

```
[113] MONA
[114] →ERR/LE+1
[115] I←W⍳W[I-1]
[116] CODE←¯2
[117] →(LOCΔ[2]≠2)/4+⎕LC
[118] NOΔ←LOCΔ
[119] NOIN←LOCIN
[120] →RETOUR
[121] MONA
[122] →ERR/LE+1
[123] →RETOUR
[124]FNI:→(CODE=0)/SYN
[125] →(CODE≠¯3)/2+⎕LC
[126] CODE←RCODE
[127] R←THEA LOCIN[1;]
[128] RE←VALΔ
[129] →(1=⍴VALΔ)/4+⎕LC
[130] REIN←VALIN
[131] →(VALΔ[2]≠3)/2+⎕LC
[132] RESD←VALSD
[133] PROC NAME
[134] →(1≠⍴VALΔ)/3+⎕LC
[135] LOCΔ←LOCIN←,0
[136] →COU
[137] LOCΔ←VALΔ
[138] LOCIN←VALIN
[139] →(LOCΔ[2]≠3)/2+⎕LC
[140] LOCSD←VALSD
[141]COU:VALΔ←REΔ
[142] →(1=⍴VALΔ)/4+⎕LC
[143] VALIN←REIN
[144] →(VALΔ[2]≠3)/2+⎕LC
[145] VALSD←RESD
[146] →ERR/LE+1
[147] →BOUCLE
[148]OPER:→(∨/CODE=0 ¯3)/2+⎕LC
[149] →SYN
[150] →(CODE=0)/NOR
[151] CODE←RCODE
[152] I←W⍳W[I]+1
[153] MONA
[154] →ERR/LE+1
[155] I←W⍳W[I-1]
[156]NOR:CODE←OPR[I⍳V[I]
[157] →(ADIC[CODE]=0 2)/ATT,RETOUR
[158] MONA
[159] →ERR/LE+1
[160] →RETOUR
[161]ATT:RCODE←CODE
[162] CODE←¯3
[163] →RETOUR
[164]BOUCLE:→(1=⍴LOCΔ)/2+⎕LC
[165] →(LOCΔ[1]=1)/FONC
[166] →(0=CODE)/SYN
[167] →(CODE≠¯3)/2+⎕LC
[168] CODE←RCODE
[169] →(CODE≠¯1)/FIL
```

```
[170] VAL∆←LOC∆
[171] VALIN←LOCIN
[172] →(1=ρLOC∆)/3+⎕LC
[173] →(3≠LOC∆[2])/2+⎕LC
[174] VALSD←LOCSD
[175] PRI←1
[176] CODE←0
[177]RETOUR:→(I=1)/3+⎕LC
[178] I←I-1
[179] →INIT
[180] →(CODE≠¯3)/4+⎕LC
[181] CODE←RCODE
[182] MONA
[183] →ERR/LE+1
[184] →(CODE≠0)/SYN
[185] LOC∆←VAL∆
[186] LOCIN←VALIN
[187] →(1=ρVAL∆)/3+⎕LC
[188] →(3≠VAL∆[2])/2+⎕LC
[189] LOCSD←VALSD
[190] →0
[191]MIL:→(CODE=¯2 21)/(2+1↑⎕LC),AFFE
[192] →(CODE≤0)/SYN
[193] DROP
[194] →ERR/LE+1
[195] →RETOUR
[196]AFFE:→(1≠ρLOC∆)/SYN
[197] →(0=ρLOCIN)/SYN
[198] →(1≠ρVAL∆)/3+⎕LC
[199] I←W⍳W[I]
[200] →NOTAF+1
[201] →(0≠LOC∆)/IND
[202] ⍞'∆',LOCIN,'←',⍕VAL∆
[203] ⍞'IN',LOCIN,'←VALIN'
[204] →(3≠VAL∆[2])/2+⎕LC
[205] ⍞'SD',LOCIN,'←VALSD'
[206] CODE←0
[207] PRI←0
[208] →RETOUR
[209]QUAD:→(CODE≠21)/INQU
[210] →(1=ρVAL∆)/NOTAF+1
[211] LOC∆←VAL∆
[212] LOCIN←VALIN
[213] →(1≠ρLOC∆)/3+⎕LC
[214] →(3≠LOC∆[2])/2+⎕LC
[215] LOCSD←VALSD
[216] PRINT
[217] PRI←CODE←0
[218] →RETOUR
[219]INQU:VV←,⎕
[220] →QUO+4
[221]CROF:→(W[I]=0)/SYN
[222] J←(⍴VV)⊤CIN RV\(RV←IQUO VV)/VV
[223] ANAL J↓¯1↓VV
[224] →(ERR=0)/3+⎕LC
[225] J←(W⍳W[I])+J-1
[226] →PART+4
```

```
[227] →(0=ρLOCΔ)/NOTAF+1
[228] →(∨/LOCΔ[1 2]≠ 0 1)/SYN
[229] →(I=ρV)/4+⎕LC
[230] I←I+1
[231] →((V[I]='←')∧~VV[J-1]∈ALCHI)/SYN
[232] I←I-1
[233] →(J≠1)/3+⎕LC
[234] J←¯1←W↓V[I]
[235] →SYN
[236] REΔ←LOCΔ
[237] REIN←LOCIN
[238] →(VV[J-1]=')')/EXPR
[239] →(VALER RV←(J-1)↑VV)/3+⎕LC
[240] I←J-1
[241] →NOTAF+1
[242] LOCΔ←,⍕'Δ',RV
[243] LOCIN←⍕'IN',RV
[244] →(3=LOCΔ[2])/2+⎕LC
[245] LOCSD←⍕'SD',RV
[246] COMU:→(0=ρLOCΔ)/⎕LC-4
[247] →(0=LOCΔ[1])/3+⎕LC
[248] I←J-1
[249] →SYN
[250] →(LOCΔ[3]=1)/4+⎕LC
[251] MSERR←'RANK ERROR'
[252] I←J-1
[253] →LE
[254] →(∧/REIN≤LOCΔ[4])/4+⎕LC
[255] MSERR←'LENGTH ERROR'
[256] I←J-1
[257] →LE
[258] →(I=ROV)/2+⎕LC
[259] →(V[I+1]='←')/AFIN
[260] →(3=LOCΔ[2])/5+⎕LC
[261] LOCIN←,LOCIN[REIN]
[262] LOCΔ← 0 1 1 ,ρLOCIN
[263] I←W↓V[I]
[264] →BOUCLE
[265] LOCIN←OTR LOCIN[,REIN;;]
[266] LOCΔ← 0 3 1 ,(ρLOCIN)[1]
[267] LOCSD←,LOCSD[REIN]
[268] I←W↓V[I]
[269] →BOUCLE
[270] EXPR:ANAL 1↓(J-2)↑VV
[271] →(ERR=0)/3+⎕LC
[272] I←J-1
[273] →PARF+3
[274] →COMU
[275] AFIN:LOCΔ←,1
[276] ADAF←REIN
[277] LOCIN←(J-1)↑VV
[278] I←W↓V[I]
[279] →BOUCLE
[280] IND:→(1=VALΔ[1])/SYN
[281] →(VALΔ[2]=⍕'Δ',LOCIN,'[2]')/3+⎕LC
[282] I←W↓V[I]+1
[283] →BOUCLE
```

```
[284] →((ρVALIN)=ρADAF)/4+⎕LC
[285] I←1+W⍳W[I]+1
[286] MSERR←'LENGTH ERROR'
[287] →LE
[288] →(3=VALΔ[2])/SPSD
[289] ⍎'IN',LOCIN,'[',(⍕ADAF),']←VALIN'
[290] CODE←0
[291] PRI←0
[292] →RETOUR
[293]DOER:MSERR←'DOMAIN ERROR'
[294] →LE
[295]SPSD:T←⍎'IN',LOCIN
[296] RESIN← 0 0 0 ρ' '
[297] J←1
[298] →(J∈ADAF)/3+⎕LC
[299] RESIN←RESIN PIL T[J;;]
[300] →2+⎕LC
[301] RESIN←RESIN PIL VALIN[ADAF⍳J;;]
[302] →(J=(ρT)[1])/3+⎕LC
[303] J←J+1
[304] →SPSD+3
[305] ⍎'IN',LOCIN,'←RESIN'
[306] ⍎'SD',LOCIN,'[',(⍕ADAF),']←VALSD'
[307] CODE←0
[308] PRI←0
[309] →RETOUR
[310]VASY:VV←1↓VV
[311] →(2≠ρVV)/SYN
[312] →(3=J←(VV∧.=MVA)⍳1)/SYN
[313] →(I=ρV)/2+⎕LC
[314] →(V[I+1]='←')/⍎'VA',⍕J
[315] →⍎'VN',⍕J
[316]VA1:→(1=ρVALΔ)/SYN
[317] →(VALΔ[1]=1)/SYN
[318] →(VALΔ[2]≠2)/SYN
[319] →(' '∈VALIN)/3+⎕LC
[320] VALIN←VALIN,' '
[321] VALΔ[3 4]←1,VALΔ[4]+1
[322] VOCA←VALIN
[323] →COVA
[324]VA2:→(1=ρVALΔ)/SYN
[325] →(VALΔ[1]=1)/SYN
[326] →(VALΔ[2]≠2)/SYN
[327] SUIV←VALIN
[328] →COVA
[329]COVA:I←W⍳W[I]
[330] CODE←PRI←0
[331] →RETOUR
[332]VN1:LOCIN←VOCA
[333] LOCΔ← 0 2 1 ,ρLOCIN
[334] →COVN
[335]VN2:LOCIN←SUIV
[336] LOCΔ← 0 2 1 ,ρLOCIN
[337] →COVN
[338]COVN:I←W⍳W[I]
[339] →BOUCLE
```

```
    ∇ DFOP
[1]    →(1=⍴LOCΔ)/VER
[2]    →(1≠⍴VALΔ)/NORM
[3]    I←W⍳W[I]
[4]   VER:MSERR←'VALUE ERROR'
[5]    ERR←1
[6]    →0
[7]   NORM:→(CODE=¯2)/FODE
[8]    →⍎'S',⍕VLA1[CODE]
[9]   S1:→((LOCΔ[2]≠1)∨(VALΔ[2]≠1))/DOE
[10]   →LEICO/LEER
[11]   →((CODE=11)∧0∈VALIN)/DOER
[12]   VALIN←⍎'LOCIN',OPRI[CODE],'VALIN'
[13]   VALΔ[3]←VAL⌈Δ3]∨LOCΔ[3]
[14]   VALΔ[4]←VAL⌊Δ4]⌈LOCΔ[4]
[15]   →FID
[16]  DOE:I←W⍳W[I]
[17]   ERR←1
[18]   MSERR←'DOMAIN ERROR'
[19]   →0
[20]  S2:→((LOCΔ[2]≠1)∨VALΔ[2]≠1)/DOE
[21]   →LEICO/LEER
[22]   →((~∧/LOCIN∈ 0 1)∨~∧/VALIN∈ 0 1)/DOE
[23]   →NORM+5
[24]  S3:→LEICO/LEER
[25]   VALΔ[2]←1
[26]   →(LOCΔ[2]=VALΔ[2])/S1+3
[27]   →((3≠LOC⌈Δ2])∧3≠VALΔ[2])/S1+3
[28]   →(LOCΔ[2]=VALΔ[2])/5+⌈LC
[29]   VALΔ[3]←VALΔ[3]∨LOCΔ[3]
[30]   VAL⌊Δ4]←VALΔ[4]⌈LOCΔ[4]
[31]   VALIN←VALΔ[4]⍴0
[32]   →FID
[33]  S4:→(LOCIN<0)/DOE
[34]   →((LOCΔ[2]≠1)∨LOC⌈Δ4]≠1)/DOE
[35]   VALΔ[3]←1
[36]   VALΔ[4]←LOCIN
[37]   →(3=VALΔ[2])/DOE
[38]   VALIN←⍎'LOCIN⍴VALIN'
[39]   →FID
[40]  S5:→((LOCΔ[2]≠2)∨VALΔ[2]≠2)/DOE
[41]   VALIN←LOCIN FSUB VALIN
[42]   VALΔ← 1 1 1 1
[43]   →ERR/0
[44]   →FID
[45]  FODE:→(2=NOΔ[4])/FODL
[46]   ARGΔ←LOCΔ
[47]   ARGIN←LOCIN
[48]   →(3≠LOCΔ[2])/2+⌈LC
[49]   ARGSD←LOCSD
[50]   ARG1Δ←VALΔ
[51]   ARG1IN←VALIN
[52]   →(3≠VALΔ[2])/2+⌈LC
[53]   ARG1SD←VALSD
[54]   NOF←⌈FX NOIN
[55]   ⍎NOF←(¯1+NOIN[1;]⍳';')↑NOIN[1;]
```

```
[56]   R←⎕EX 'NOF'
[57]   →ERR/0
[58]   VALΔ←RESΔ
[59]   VALIN←RESIN
[60]   →(3≠VALΔ[2])/FID
[61]   VALSD←RESSD
[62]   →FID
[63] FODL:R←THEA 'NOIN'[1;]
[64]   PROC NAMF
[65]   →FID
[66] FID:CODE←0
[67]   PRI←1
[68]   →0
[69] LEER:ERR←1
[70]   MSERR←'LENGTH ERROR'
[71]   →0
[72] S12:→((LOCΔ[2]≠1)∧LOCΔ[4]≠0)/DOE
[73]   →(~∧/LOCIN∈ 0 1)/DOE
[74]   →((LOCΔ[4]=1)∨VALΔ[4]=1)/2+⎕LC
[75]   →(LOCΔ[4]≠VALΔ[4])/DOE
[76]   →(VALΔ[4]=1)/REPT
[77]   VALΔ[4]←⍴AIN←LOCIN/⍳VALΔ[4]
[78]   →(VALΔ[2]=3)/3+⎕LC
[79]   VALIN←VALIN[AIN]
[80]   →FID
[81]   VALIN←VALIN[AIN;;]
[82]   VALSD←VALSD[AIN]
[83]   →FID
[84] REPT:VALΔ[4]←+/LOCIN
[85]   →(VALΔ[2]=3)/3+⎕LC
[86]   VALIN←VALΔ[4]⍴VALIN
[87]   →FID
[88]   VALSD←VALΔ[4]⍴VALSD
[89]   VALIN←((VALΔ[4]×(⍴VALIN)[1]),(⍴VALIN)[2 3])⍴VALIN
[90]   →FID
[91] S6:→(LOCΔ[2]=VALΔ[2])/⎕LC+4
[92]   VALΔ← 0 1 ,LOCΔ[3 4]
[93]   VALIN←VALΔ[4]⍴0
[94]   →FID
[95]   →(3=LOCΔ[2])/DOE
[96]   VALΔ← 0 1 ,LOCΔ[3 4]
[97]   VALIN←LOCIN∈VALIN
[98]   →FID
[99] S7:→(LOCΔ[2]≠1)/DOE
[100] →(LOCΔ[4]≠1)/DOE
[101] →(VALΔ[2]=3)/7+⎕LC
[102] →(CODE=17)/3+⎕LC
[103] VALIN←LOCIN↓VALIN
[104] →2+⎕LC
[105] VALIN←LOCIN↓VALIN
[106] VALΔ[3 4]←1,⍴VALIN
[107] →FID
[108] AIN←⍋(⍒LOCIN),OPRI[CODE],⍒⍳VALΔ[4]
[109] VALΔ[3 4]←1,⍴AIN
[110] VALIN←VALIN[AIN;;]
[111] VALSD←VALSD[AIN]
[112] →FID
```

```
[113]S8:→(LOCΔ[2]=VALΔ[2])/4+⎕LC
[114] VALΔ← 0 1 ,LOCΔ[3 4]
[115] VALIN←VALΔ[4]ρ0
[116] →FID
[117] →(3=LOCΔ[2])/DOE
[118] VALΔ[1 2]← 0 1
[119] VALIN←LOCIN⍳VALIN
[120] →FID
[121]S13:→((LOCΔ[2]≠2)∨(LOCΔ[4]≠1)∨VALΔ[2]≠2)/DOE
[122] VALIN←LOCIN FABS VALIN
[123] VALΔ← 1 1 1 1
[124] →ERR/0
[125] →FID
[126]S15:→(LOCΔ[4]≠0)/3+⎕LC
[127] VALΔ[3]←1
[128] →FID
[129] →(VALΔ[4]≠0)/5+⎕LC
[130] VALΔ←LOCΔ
[131] VALΔ[3]←1
[132] VALIN←LOCIN
[133] →FID
[134] →(LOCΔ[2]≠VALΔ[2])/DOE
[135] VALΔ[3]←1
[136] VALΔ[4]←VALΔ[4]+LOCΔ[4]
[137] →(LOCΔ[2]=3)/3+⎕LC
[138] VALIN←LOCIN,VALIN
[139] →FID
[140] AIN←(ρLOCIN)[2 3]⍴(ρVALIN)[2 3]
[141] VALIN←(((ρLOCIN)[1],AIN)↑LOCIN),[1]((ρVALIN)[1],AIN)↑VALIN
[142] VALSD←LOCSD,VALSD
[143] →FID
[144]S16:→(0=VALΔ[4])/FID
[145] →((VALΔ[2]≠2)∨LOCΔ[2]≠3)/DOE
[146] →(∨/LOCSD≠0)/DOE
[147] →(∧/VALIN∈VOCA)/FID
[148] R←(~VALIN∈VOCA)/VALIN
[149] R←R[((⍳ρR)∈R⍳R)/⍳ρR]
[150] IK←1
[151] R1←⍳0
[152] →(VALIN[IK]∈VOCA)/⎕LC+4
[153] →((IL←R⍳VALIN[IK])>(ρLOCIN)[1])/3+⎕LC
[154] R1←R1,SBF LOCIN[IL;1;]
[155] →2+⎕LC
[156] R1←R1,VALIN[IK]
[157] IK←IK+1
[158] →(IK≤ρVALIN)/¯6+⎕LC
[159] VALIN←R1
[160] VALΔ[3 4]←1,ρR1
[161] →FID
[162]S9:→((LOCΔ[2]=1)∨VALΔ[2]=1)/DOE
[163] →⍎'S9A',(⍕LOCΔ[2]),⍕VALΔ[2]
[164]S9A22:VALSD←,1
[165] VALIN←(1,ρR)ρR←((1,ρLOCIN)ρLOCIN) LIS VALIN
[166] VALΔ← 0 3 0 1
[167] →FID
```

```
[168]S9A32:→(LOCΔ[4]>1)/DOE
[169] R←LOCIN[1;;]
[170] R←R LIS VALIN
[171] →(LOCSD=(ρR)[1])/3+⌈LC
[172] R1←(LOCSD,0)↓  1 0 ↓R
[173] VALIN←(1,ρR)ρR
[174] VALΔ← 0 3 0 1
[175] VALSD←LOCSD+1
[176] →FID
[177]S9A23:→(VALΔ[4]>1)/DOE
[178] R←VALIN[1;;] LIS LOCIN
[179] VALIN←(1,ρR)ρR
[180] →FID
[181]S9A33:→DOE
[182]S11:→(0=VALΔ[4])/FID
[183] →(2≠VALΔ[2])/DOE
[184] →((1≠LOCΔ[2])∨LOCΔ[4]>1)/DOE
[185] →(~LOCIN∈0,⍳2)/DOE
[186] →⍎'S11',⍕LOCIN
[187]S111:VALIN←SBI VALIN
[188] VALΔ[3 4]←1,ρVALIN
[189] →FID
[190]S112:VALIN←SBF VALIN
[191] →S111+1
[192]S110:VALIN←((VALIN≠' ')∨VALIN≠1⌽VALIN)/VALIN
[193] VALIN←SBF VALIN
[194] →S111+1
     ∇
```

```
     ∇ MONA;NOF;V
[1]    →(1=ρLOCΔ)/SYN
[2]    →(1=ρVALΔ)/SYN
[3]    →(CODE=¯2)/MFOD
[4]    →⍎'S',⍕VLA2[CODE]
[5]   S1:→(VALΔ[2]≠1)/DOE
[6]    VALIN←@PRI[CODE],'VALIN'
[7]    →FIM
[8]   S2:→(VALΔ[2]≠1)/DOE
[9]    →(~∧/VALIN∈ 0 1)/DOE
[10]   VALIN←~VALIN
[11]   →FIM
[12]  S3:VALIN←VALΔ[3]/VALΔ[4]
[13]   VALΔ[2 3 4]←1,1,ρVALIN
[14]   →FIM
[15]  S4:→((VALΔ[2]≠1)∨VALΔ[4]≠1)/DOE
[16]   →(VALIN<0)/DOE
[17]   VALΔ[3 4]←1,VALIN
[18]   VALIN←⍳VALIN
[19]   →FIM
```

```
[20] S13:→(VALΔ[2]≠3)/DOM
[21]   →(1<ρVALSD)/DOM
[22]   VALIN←,VALIN[1;VALSD+1;]
[23]   VALIN←φ(¯1+(' '=φVALIN)ι0)↓φVALIN
[24]   VALΔ← 0 2 1 ,ρVALIN
[25]   →FIM
[26] S5:→(VALΔ[2]≠2)/DOM
[27]   VALIN←(1 1 ,ρVALIN)ρVALIN
[28]   VALSD←,0
[29]   VALΔ← 0 3 1 1
[30]   →FIM
[31] FIM:CODE←0
[32]   PRI←1
[33]   →0
[34] DOM:ERR←1
[35]   MSERR←'DOMAIN ERROR'
[36]   →0
[37] METOD:→(2=LOCΔ[4])/FOLD
[38]   ARGΔ←VALΔ
[39]   ARGIN←VALIN
[40]   →(3≠VALΔ[2])/2+⎕LC
[41]   ARGSD←VALSD
[42]   NOM←⎕EX LOCIN
[43]   ⍎NOM←(¯1+LOCIN[1;]ι';')↑LOCIN[1;]
[44]   R←⎕EX NOM
[45]   →ERR/0
[46]   VALΔ←RESΔ
[47]   VALIN←RESIN
[48]   →(3≠VALΔ2])/FIM
[49]   VALSD←RESSD
[50]   →FIM
[51] FOLD:R←THEA LOCIN[1;]
[52]   PROC NAME
[53]   →FIM
[54] S10:→(3≠VALΔ2])/DOM
[55]   →(0=ρVALSD)/NIL
[56]   →(1≠∧/VALSD[1]=VALSD)/DOM
[57]   AD←VALSD[1]+1
[58]   →(AD>3)/DOM
[59]   →(AD=3)/DY
[60]   VALIN←FPAS VALIN
[61]   VALΔ← 1 1 1 1
[62]   →ERR/0
[63]   →FIM
[64] DY:VALIN←FDAS VALIN
[65]   VALΔ← 1 2 1 1
[66]   →ERR/0
[67]   →FIM
[68] NIL:
[69] S12:→(∨/VALΔ[2 4]≠ 2 2)/DOM
[70]   →(VALIN[1]=VALIN[2])/DOM
[71]   VALIN←FTPP VALIN
[72]   VALΔ← 1 1 1 1
[73]   →FIM
```

```
[74] S11:VALΔ[3]←1
[75]   →FIM
[76] S8:→(VALΔ[4]=0)/VID
[77]   →(VALΔ[2]≠2)/DOE
[78]   V←VALIN
[79]   SSB
[80]   ANAL V
[81]   →ERR/DOE
[82]   →(1=ρLOCΔ)/VID
[83]   VALΔ←LOCΔ
[84]   VALIN←LOCIN
[85]   →(3≠LOCΔ[2])/2+⌈LC
[86]   VALSD←LOCSD
[87]   →FIM
[88] VID:VALΔ←VALIN←,0
[89]   →FIM
[90] S9:→(VALΔ[2]≠1)/FIM
[91]   VALΔ[2 3]← 2 1
[92]   VALIN←⍕VALIN
[93]   VALΔ[4]←ρVALIN
[94]   →FIM
```