MASTER EN SCIENCES INFORMATIQUES

Une implémentation de LUCID

Gardin, Patrick

Award date:
1978

Awarding institution:
Universite de Namur

FACULTÉ UNIVERSITAIRES NOTRE-DAME DE LA PAIX
NAMUR
INSTITUT D'INFORMATIQUE

Année académique 1977-1978

UNE IMPLÉMENTATION DE

LUCID

Patrick GARDIN

Mémoire présenté en vue de l'obtention
du grade de
Licencié et Maître en Informatique.

FACULTÉ UNIVERSITAIRES NOTRE-DAME DE LA PAIX
NAMUR
INSTITUT D'INFORMATIQUE

Année académique 1977-1978

UNE IMPLÉMENTATION DE

LUCID

Patrick GARDIN

Mémoire présenté en vue de l'obtention
du grade de
Licencié et Maître en Informatique.

Nous remercions les membres du département d'informatique de l'Université de Warwick pour leur accueil.
Nous tenons à remercier tout spécialement Mr Wadge et ses amis. Leur soutien et leurs conseils ont été une aide précieuse dans l'élaboration de ce projet.
Qu'il me soit également permis d'exprimer mon entière gratitude à Mr Cherton dont les conseils et critiques ont facilité la rédaction.

TABLE DES MATIERES

AVANT-PROPOS

De plus en plus de littérature paraît sur les techniques "de démonstration de programmes". [ACM - ACTA INFORMATICA ...]
Pour arriver à leur fin, les auteurs suivent tous le même chemin : ils essayent de greffer des "outils mathématiques" sur des outils informatiques. Partant de certaines particularités de programmes ou de langages, ils essayent d'écrire un modèle mathématique dans lequel il est possible de traduire le programme et de démontrer ainsi qu'il est correct. Actuellement, ces modèles sont généralisés ou de nouveaux modèles sont écrits de telle sorte qu'ils puissent être adaptés à toute une variété de programmes.

Lucid, bien qu'ayant le même but, diffère de ces méthodes par sa façon de résoudre le problème. Etant donné le nombre d'années et le nombre de chercheurs travaillant dans le domaine cité plus haut, nous pouvons déduire qu'il n'est pas évident d'adapter les mathématiques à l'informatique. Le processus inverse est peut être plus efficace, et c'est dans cette direction que s'est développé Lucid.

Le langage Lucid sert à la fois à exprimer le modèle à l'aide duquel on vérifie un programme et les programmes eux-mêmes. Plutôt que de partir de choses existantes, les auteurs ont construit un modèle répondant à leurs besoins

- pouvoir démontrer, dans celui-ci, qu'un programme est correct
- pouvoir écrire un programme dans ce modèle.

Le modèle mathématique et l'outil informatique ne font plus qu'un en Lucid. Un programme Lucid est à considérer comme un ensemble d'équations vraies à tout instant. Ainsi si nous nous avons l'équation :

X = A + B

alors "X" sera toujours égal à "A + B" quelle que soient les valeurs de "A" et de "B". Nous pouvons ainsi remplacer toute occurrence de "X" par "A + B".
Un programme Lucid est aussi un ensemble d'équations définissant des variables. L'output du programme sera défini en fonction de ces variables. Le programmeur n'a donc pas à se soucier de la façon dont l'output est évalué. Si l'on considère une équation comme une commande, nous ne pouvons plus déterminer dans quel ordre ces commandes seront exécutées. Le caractère séquentiel a donc disparu dans un programme Lucid. Il est cependant toujours possible d'exprimer des événements se déroulant dans le temps. Ainsi si nous avons :

X = A asa B

alors "X" prendra la valeur qu'aura "A" dès que "B" est "VRAI". Quand cette condition sera-t-elle vérifiée ? Tout dépend de la définition de "B".
La suppression du séquentiel et le maintien de la notion de temps permet ainsi de décrire de façon plus naturelle des algorithmes : il ne faut plus se soucier de la façon dont vont se dérouler les événements en machine.

Remarques

- Nous omettrons, dans ce rapport, tout ce qui concerne le formalisme Lucid, par manque de connaissances dans ce domaine.
- Abréviations utilisées
  - asa as soon as
  - cmp compute
  - fby followed by
  - fun function
  - map mapping
  - pro produce
  - usg using
- mots clés
  - Basic Lucid : Lucid défini dans [1]
    Basic Lucode : Lucode correspondant au Basic Lucid
  - Lucid étendu : Basic Lucid + clauses
    Lucode étendu : Lucode correspondant au Lucid étendu
- Nous nous excusons pour l'emploi de certains mots tel que :
  . appel en fonction du besoin pour "call by need"
  . transformateur pour "transformer"
  . rebaptiser pour "rename".

CHAPITRE I

LUCID ETENDU

§1. LA STRUCTURE DE "BLOC" EN LUCID

Une structure souvent utilisée dans les langages de programmation de haut niveau (Algol, PL/1, ...) est la structure de bloc. Elle permet de désigner plusieurs objets par un même nom. Cette structure avait été introduite en Basic Lucid au moyen des délimiteurs "begin" et "end". [1]

Particularités du "begin-end".

- Nous définissons "variables globales" comme étant les variables définies en dehors des parenthèses "begin-end".
  L'emploi de ces variables à l'intérieur d'un bloc ainsi défini signifie que l'on utilise la valeur qu'elles avaient au moment de rentrer dans le bloc.(*) Lors de chaque itération à l'intérieur d'une boucle ordinaire (non dans un bloc "begin-end") nous prenions une nouvelle valeur pour chacune des variables utilisées. Nous nous servions donc de l'historique de toutes les variables utilisées. Par contre dans le cas d'une boucle à l'intérieur des délimiteurs "begin-end", nous utilisons toujours la même valeur, pour les variables globales, quelle que soit le nombre d'itération à l'intérieur de la boucle.
- La variable "interface" (définie dans le bloc, et utilisée en dehors de celui-ci) est telle qu'elle ne fournit de valeur que lors de chaque passage dans le bloc. (suite à une demande) Les variables locales sont réinitialisées lors de chaque traversée du bloc.

Les auteurs de Lucid ont généralisé cette structure et introduit la notion de fonction. Les nouvelles constructions introduites en Lucid sont appellées "clauses".

Etant donné la non parution, à ce jour, de l'article traitant ce sujet, nous nous permettons d'en faire un bref "résumé".

Note : Signalons que la structure "begin-end" a été remplacée par une des clause introduites, nous y reviendrons.

(*) Note importante :

- temps = temps Lucid définit dans [1]
- nous "rentrons dans un bloc" lorsqu'une équation fait référence au nom de ce bloc. (demande d'évaluation de l'output du bloc)

1. Clause "PRODUCE"

Comme toutes les clauses qui seront définies, elle permet de limiter la portée de définition de certaines variables.

Format d'une clause produce

produce <data term> using <variable list>
<set of assertions>
end

où : <data term> = sujet de la clause (ou nom de la clause dans ce rapport)
<variable list> = liste des variables globales
<set of assertions> = corps de la clause.

Les variables définies à l'intérieur du corps de la clause et en particulier la variable "output" sont appelées variables locales.

Le corps d'une clause produce définit le sujet de celle-ci (par l'intermédiaire de la variable spéciale "output") en fonction des variables globales et locales. (Celles-ci peuvent être également des clauses).
Malgré l'introduction de la structure "begin-end" en 1 , nous considérons que la clause produce est le premier prolongement logique de Lucid : au lieu de définir une variable au moyen d'une équation, nous la définissons au moyen d'un ensemble d'équations. La variable output d'un programme Basic Lucid était d'ailleurs déjà définie de la sorte. Un programme en Basic Lucid est donc à considérer comme une clause produce.

Une équation du Basic Lucid peut être considérée à la limite comme une clause produce où

- le sujet = partie gauche de l'équation
- les variables globales = ensemble des variables utilisées dans la partie droite
- l'ensemble des assertions se résume à une seule : l'assertion existant en Basic Lucid mais dont on a remplacé la partie gauche par la variable "OUTPUT".

2. Clause FUNCTION

Cette clause est analogue à PRODUCE : plutôt que de définir une variable, elle définit une variable fonction.

Format d'une clause function

function <function term> (<data variable list>) using <variable list>
<set of assertions>
end

<function term> = sujet
<data variable> = liste des paramètres formels
<variable list> = liste des variables globales

Une variable est locale si - elle apparaît dans le corps de la clause
- elle n'est ni variable globale, ni paramètre formel.

> *"A function clause is an assertion about the subject and globals. It asserts that, for all values of the formal parameters, there exist values of the local variables which make every assertion in the body of the clause true when "output" has the value of the subject applied to the formal parameters values."*[3]

Ces deux types de clauses ont la particularité de mémoriser la valeur de leurs variables locales entre deux accès successifs.

3. Clause "COMPUTE"

Il s'agit d'un cas particulier de la clause PRODUCE. Elle remplace la structure "begin-end" introduite en [1]

La forme est semblable à celle du PRODUCE : "compute" remplace "produce"
"result" remplace "output"

4. Clause MAPPING

Il s'agit d'un cas particulier de la clause FUNCTION. Sa forme en est d'ailleurs semblable : "mapping" au lieu de "function"
"result" au lieu de "output"

Fonction de ces deux dernières clauses

Elles ont pour effet de geler leurs variables globales - et paramètres formels dans le cas de "mapping" - : quelle que soit le nombre d'itérations dans le corps d'une de ces clauses, on utilise toujours la valeur qu'avaient les variables globales - les paramètres - au moment d'entrer dans la clause. Les valeurs des variables locales ne sont plus mémorisées entre deux accès successifs au corps de la clause. Définies en fonction des variables globales, elles sont réinitialisées lors de chaque nouvel accès à la clause.

Les clauses compute - mapping sont respectivement équivalentes aux clauses produce - function si on applique l'opérateur latest [1]

- aux variables globales
- aux paramètres formels dans le cas de mapping et l'opérateur latest^{-1}
- à la variable "result" (result $\simeq$ latest output)

Considérons le programme des nombres premiers [1]

```
n = ...
i = 2 fby i+1
compute idivn using i, n
   j=i*i fby j+i
   result = j eq n asa    j not < n
end
output = not idivn   asa    idivn or i*i not < n
```

Ce programme est équivalent à :

```
n = ...
i = 2 fby i+1
produce idivn using i, n
   j = latest i * latest i fby j + latest i
   output = j eq latest n asa j not < latest n
end
output = not idivn  asa  idivn  or i*i not < n
```

Latest appliqué à n'importe quoi doit être stationnaire [4] : la valeur ne peut changer à l'intérieur d'une boucle. Le résultat d'un "compute" ou "mapping" doit donc être une constante. Ceci est assuré aux moyens d'expressions de la forme : X asa P

Si l'on devait comparer avec Algol, nous dirions qu'un compute est similaire à un bloc n'ayant pas de "side effect".

Mode de fonctionnement

1. Compute-mapping

Les variables globales - les paramètres formels - sont considérés comme constant pendant toute la durée de l'évaluation du sujet de la clause.

2. Produce - function

Le mécanisme est tout autre : il n'y a plus de gel des variables globales ni des paramètres éventuels.

> *"Function clauses can be thought of as templates for processes, with each textual occurrence of a function call corresponding to the process which is the appropriate instance of the template. These processes must, like those defined by produce clauses, be thought of as operating in parallel, but synchronized with the enclosing iteration, and as updating internal variables even if, on some steps of the enclosing iteration, the output values are not required.*

Alternatively, the function body can be thought of as conventional Algol - like procedure body which is called and returns a result, provided in addition that

(i) the inductively defined local variables are thought of as own variables whose values are remembered between one call and the next;

(ii) different textual occurrences have separate copies of these variables; and

(iii) the procedure is called on each step of the iteration containing the function call, even when the value is not needed, for "housekeeping purposes", namely to keep the own variables up to date."[3]

Etant donné la dernière particularité, nous dirons que ces deux types de clauses sont des processus itératifs : ils restent actifs pour permettre la mise à jour de leurs variables locales.

exemple : [3]

```
produce y using x
    n = 1 fby n + 1
    t = first x fby t + next x
    output = t/n
end
```

- La troisième valeur de "y" sera la moyenne des trois premières valeurs de "x".

Tableau résumé

	cmp	fun	map	pro
limite la portée de définition	X	X	X	X
gèle les variables globales	X		X	
gèle les paramètres actuels			X	
définit des fonctions		X	X	
Comparaison avec Algol	bloc sans side effect	pro-cedure + own	proce-dure sans side effect	bloc + own

Si l'on voulait représenter les dépendances entre variables au moyen d'un graphe [2] [chap. II] dont

- les sommets sont les variables
- les arcs représentent les dépendances temps (reflet d'une équation Lucid) entre variables, alors la représentation d'une clause constituerait un sous graphe. Ce dernier pouvant contenir d'autres sous graphes si la clause contient des définitions de clauses. Ces sous graphes sont tels que :
  (1) un sommet d'un sous graphe peut avoir 0, 1 ou plusieurs sommets ascendants. (il peut être ascendant de lui-même)
  (2) un sommet d'un sous graphe ne peut être ascendant d'un sommet en dehors de ce sous graphe.

Note : Un sommet "X" est ascendant à un sommet "Y"
si "Y" est défini en fonction de "X"
(1) traduit la notion de variable globale
(2) traduit la notion de variable locale.

Vérification des programmes

Les clauses n'enlèvent rien au but de Lucid : pouvoir vérifier mathématiquement qu'un programme est correct. Il existe en effet des règles permettant de résoudre ce problème. Une vérification est faite pour la structure "begin-end" dans [4].

L'introduction des clauses en Lucid a permis, entre autre, de définir une variable au moyen de plusieurs équations. Cette modularisation devrait nous permettre une meilleure méthodologie lors de la vérification de programme. L'introduction des fonctions est d'un intérêt parfois considérable pour la vérification de programme : des fonctions standards : écrites et vérifiées une fois pour toute, allègeront la vérification des programmes qui les utilisent.

Remarque :

Le nom des types de clause et des variables output reflète bien leur fonction.

COMPUTE X... = calculer X, c'est-à-dire lui fournir une valeur, d'où le nom de la variable de sortie : result

PRODUCE X ... = produire reflète bien le caractère itératif d'une clause. La sortie étant un flux de données, la variable de sortie est appelée output

CHAPITRE II :

UNE IMPLEMENTATION DE LUCID

§1 CHOIX DU TYPE D'IMPLEMENTATION

Trois types d'implémentation nous étaient proposés [1] :

- compilateur
- système de flux de données
- interpréteur

Rappellons [1] que les deux premiers modes d'implémentation n'acceptent pas tous les programmes Lucid.

Dans le cas du compilateur Basic Lucid [2], l'auteur construit un graphe représentant les dépendances entre variables. Il utilise les règles suivantes :

Lorsqu'une variable "A".est définie comme étant :

- first x : tracer un arc de A vers x de poids 0
- next x : tracer un arc de A vers x de poids 1
- x asa y : tracer un arc de A vers x et de A vers Y tous deux de poids ∞
- x fby y : tracer un arc de A vers x de poids 0 et un arc de A vers Y de poids -1
- toute autre formule : dresser un arc de poids 0, vers chaque variable apparaissant dans la formule

Il définit ensuite le sous-ensemble de programmes compilables comme étant l'ensemble des programmes Lucid tel que :

- toute variable référencée doit être définie une et une seule fois (excepté pour la variable input qui peut ne pas être définie (cette variable n'a pas été reprise dans notre implémentation)
- La variable output doit être définie
- La somme des poids des arêtes formant un cycle doit être négative.

Ceci revient à supprimer les programmes contenant des variables dépendant de leur propre futur, ce qui rejoint l'idée émise dans [1].

Notons que cette dépendance doit être finie, sans quoi la variable n'est pas définie.

Le même problème se pose dans le cas d'un système de flux de donnée.

Considérons l'équation :

x=if B then E_1
else E_2 (exemple fortement inspiré du programme écrit dans [1]) où E_2 est fonction de next x

Si "B" à l'instant 1 est "False" alors le système est incapable de fournir une valeur pour "x" à l'instant t=1 et donc incapable de poursuivre sa production de valeurs.

L'interpréteur par contre utilisera la définition récursive de "x" pour pouvoir en déterminer sa valeur.

L'interpréteur ou le système de flux de données nous tenaient particulièrement à coeur :

- ces deux modes d'implémentation nous étaient quelque peu inconnus;
- ils réflètent bien les dépendances entre les variables définies dans un programme Lucid.

Comparons donc, brièvement, le fonctionnement de chacune de ces deux méthodes d'implémentation dans le cadre du "Basic Lucid".

1.1. Système de flux de données

Considérons le programme qui calcule la racine carrée d'un nombre [1]

```
n = ...
i = 0 fby i+1
j = 1 fby j+(2*i)+3
out = i asa j>n
```

Ce programme peut être représenté [fig. 1] au moyen d'un graphe dont :

- les sommets sont les opérateurs
- les arcs sont les variables.

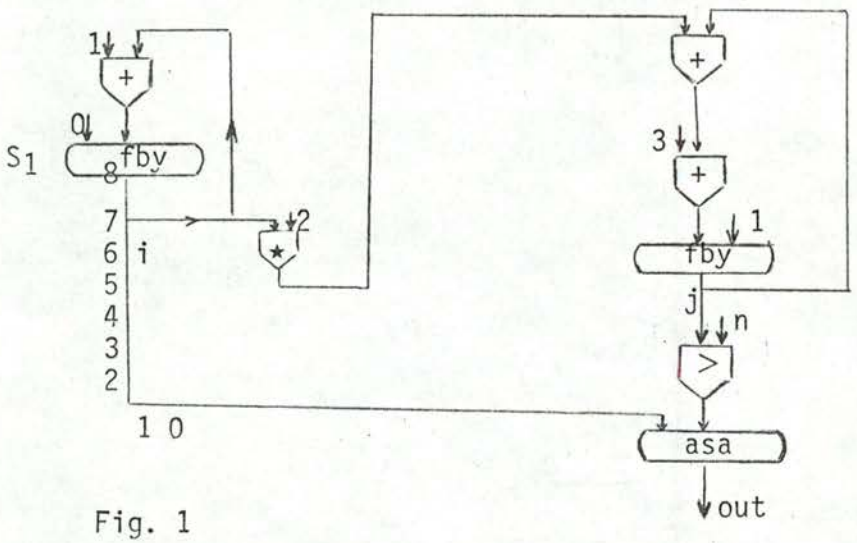


Fig. 1

Un sommet peut être considéré comme une "entreprise" produisant des valeurs pour la variable résultante.

Les sommets sont tous "activés" en même temps : au début de "l'exécution" du programme.

Les valeurs que prennent les différentes variables "circulent" dans le graphe. Lorsqu'un sommet a besoin de la valeur d'une variable à l'instant t, il prendra la valeur à sa borne d'entrée à l'instant "t".

Exemple : soit t=4 tel que $j(t) > n(t)$

$$n(t) = n(o) \ \forall t$$

Le sommet "asa" produisant les valeurs de "out" prendra à l'instant t=4, la valeur qui se trouve sur la borne "i", c'est-à-dire "3"

Un système de flux de données fonctionne avec un temps discret. Il se peut qu'un sommet soit "bloqué" (attente de valeurs d'un sommet ascendant, ex : sommet descendant d'un sommet "asa")

Le tout doit être synchronisé au moyen d'une horloge.

Soient - s un sommet

- S' l'ensemble des sommets descendants de s

Si "s" n'appartient pas à S' alors "s" "fonctionne" indépendemment de tout sommet de S'. Pendant la production d'une valeur de j, le sommet S_1 [fig. 1] peut déjà préparer la prochaine valeur de "i".

1.2. Interpréteur

Cette fois, les sommets ne sont plus activés en même temps. Nous activons le sommet S(out) qui activera le sommet S(j) qui ... (où S(j) = sommet qui alimente la variable "i").

Un sommet ne sera actif (producteur de valeurs) durant un temps infini que si son sommet ascendant adjacent l'est. L'interpréteur travaille suivant un processus "d'aspiration" tandis qu'un système de flux de données fonctionne suivant un processus "d'alimentation".

L'idée de départ étant d'implémenter le Lucid étendu et, étant donné qu'une version de l'interpréteur existait déjà, il nous a paru plus raisonnable de choisir l'interpréteur, plutôt que d'écrire un système de flux de données et de l'étendre aux clauses.

§2. VUE GENERALE DE L'INTERPERTEUR

2.1. Rappel

- Compilateur

  Le programme passe par deux stades :

  - la compilation
  - l'exécution du programme objet

- Interpréteur

  Le programme source est exécuté directement.

2.2. Notre implémentation

Un programme Lucid passera au travers de deux phases : - transformation
- interprétation

L'interface entre les deux sera un programme écrit en un code appellé Lucode.

Nous aurons donc la configuration suivante :

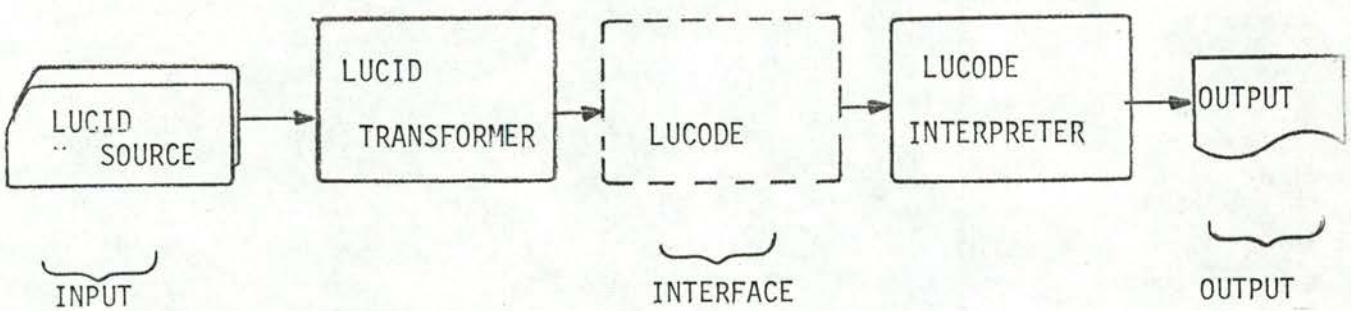


FIG. 2 : L'outil Lucid

2.3. Fonctions des deux phases

a) Le programme Lucode

Un programme Lucode est un ensemble d'équations, n'ayant chacune qu'un et un seul opérateur.

Une équation Lucode a la configuration suivante :

< variable >[=] < opérateur >< opérande 1 >< opérande 2 >

. [=] Le signe d'égalité n'est pas repris dans le représentation interne de l'équation

. L'opérande 2 est optionnel (cas des opérateurs unaires)

Représentation interne

Les variables et les opérateurs sont représentés au moyen d'un nombre. Ce nombre est porteur d'une double information :

- la catégorie (séparateur, variable ou constante)
- il spécifie de quel élément il s'agit dans cette catégorie.

b) Le "transformateur"

Il traduit le programme Lucid en un programme Lucode équivalent et plus facilement interprétable. Trois opérateurs seront introduits au cours de cette transformation :

- l'opérateur "α" (ALPHA) traduit la notion de paramètre formel
- l'opérateur "β" (BETA) traduit un appel de clause
- l'opérateur "γ" (GAMMA) traduit la notion de variable globale

Nous ferons également "exploser" une équation à plusieurs opérateurs en plusieurs équations n'ayant chacune qu'un et seul opérateur.

c) L'interpréteur

Il interprète le programme Lucode suivant le procédé énoncé en [1].

§3. LE TRANSFORMATEUR LUCID

Nous avons gardé un schéma d'analyse analogue à tout analyseur de programme source :

- analyseur lexicographique
- analyseur syntaxique

Nous y avons ajouté une troisième étape qui consiste à "rebaptiser" les variables. Ceci n'est nécessaire que dans le mesure où l'on veut accroître les performances de l'interpréteur. Le transformateur peut être considéré comme un processus se déroulant en huit étapes [fig. 4] . Nous n'avons pas repris le transformateur du Basic Lucid comme tel.

Nous y avons adopté le mécanisme de l'interpréteur existant : "appel en fonction du besoin" (call by need)

3.1. "Appel en fonction du besoin"

Soient deux routines "A" et "B"

Nous dirons que nous avons un "appel en fonction du besoin" lorsqu'un passage de paramètre se fait comme suit :

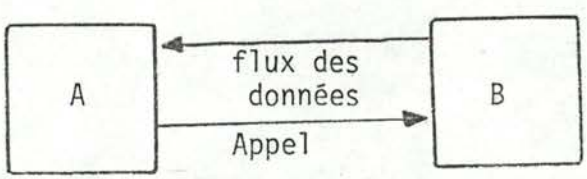
- un paramètre a une valeur non significative ou n'a pas de valeur du tout lors de l'appel de "B"
- ce paramètre est garni par "B"
- "A" se sert du paramètre après l'appel.

La routine "A" appelle "B" pour que cette dernière garnisse le(s) paramètre(s) de telle sorte que A puisse continuer son traitement.

Ce type d'appel est très pratique dans une programmation modulaire :

- le passage des valeurs des paramètres ne se fait que suivant une seule direction
- le flux des informations reflète bien la logique du problème
- il évite des appels trop imbriqués, d'où facilité de :
  - compréhension, lisibilité, ...
  - mise à jour

Représentons graphiquement ce mode d'appel :



Remarquons que pendant que "A" traite les données reçues de "B"; "B" peut déjà préparer les prochaines données qu'il aura à transmettre.

3.2. Remarque

Les huit modules [fig. 4] ne sont pas tous synchronisés de la même façon : le module "BETA" n'appellera qu'une seule fois le module "SYNAN" alors que ce dernier devra appeller "OPDRI" autant de fois qu'il y a de phrases Lucid :

< phrase Lucid >:: = <équation > / < en-tête > / < "end" >

3.3. L'analyseur lexicographique

3.3.1. Définitions

<entité >:: = <chaîne de caractères représentant une variable >/
<chaîne de caractères représentant un mot réservé >/
<chaîne de chiffres décimaux représentant un nombre >

<séparateur > :: = <caractère séparateur > / < opérateur >

<caractères séparateurs > :: =

-/♭/(/)/,/./'/!/;/:/?/@/#/&/%/$/

<opérateurs > :: =<opérateurs habituels>/<opérateurs Lucid>

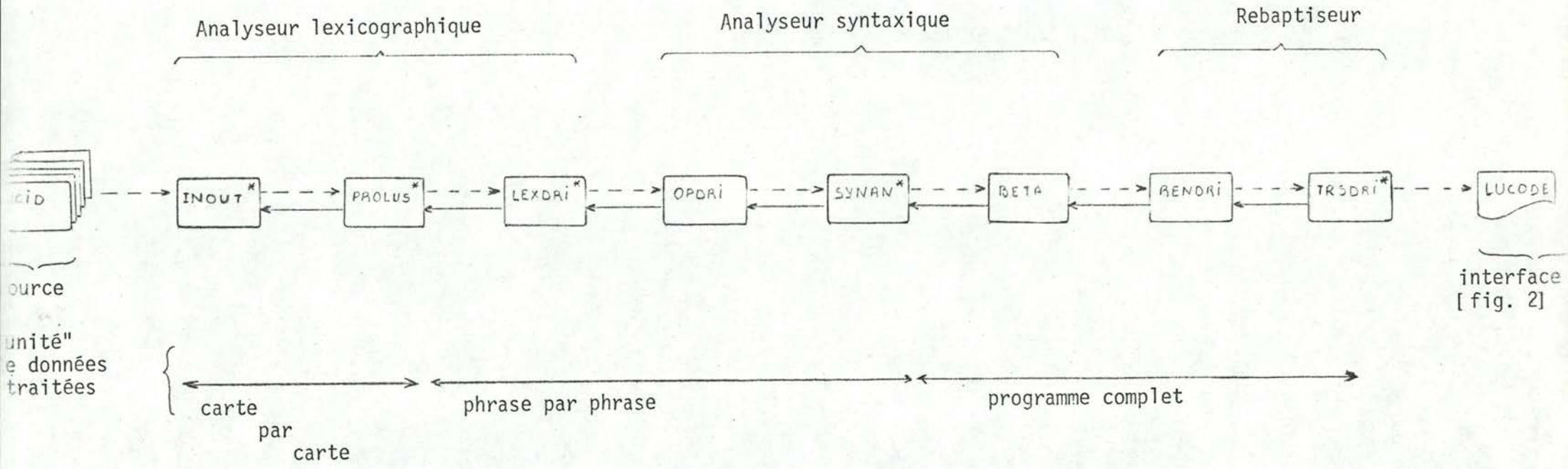
<opérateurs habituels > :: =

+/-/*///</>/not/or/eq/exp/And/pow/ln

où exp : exponentiel
pow : exposant (power)

asa	as soon as
fst	first
nxt	next
usg	using
upon	upon
when	whenever
fby	followed by

FIG.3. : Opérateurs Lucid



(*) similaire à ce qui était fait en Basic Lucid

FIG. 4. : Le transformateur Lucid

Outres les mots réservés, les seules entités qui diffèrent des entités définies en Basic Lucid sont :

- appel d'une clause à argument(s)

  ex : $f(a_1, a_2, \ldots)$

- définition d'une clause

  ex : fun f(x, y)

  Les entités sont : fun mot réservé
  f variable (nom d'une clause)
  (,) séparateurs
  x, y } variables

L'analyseur lexicographique n'a guère dû être modifié, les extensions étant faciles à réaliser dans une programmation modulaire. Les entités propres au Lucid étendu sont facilement reconnaissables : une parenthèse ouvrante précédée d'un nom de variable. Dans ce cas, l'analyseur substituera à la '(' un autre séparateur, qui n'est utilisé que par lui.

3.3.2. Configuration interne des caractères : Luscii

L'analyseur lexicographique a à sa disposition une table [fig. 5] permettant de faire la conversion de caractères en un code, appellé Luscii. Ce code permet une plus grande souplesse lors de l'identification des différentes entités.

3.3.3. Le code Lucode

Ce code est la représentation interne des différentes entités reconnues par l'analyseur lexicographique.

Code Lucode des opérateurs (séparateurs)

. s'il s'agit d'un opérateur représenté en EBCDIC par un seul caractère (ex : +, *, ...) alors ce code est le même que le code Luscii de cet opérateur

. sinon, ce code est obtenu au moyen d'une table.

Code Lucode des variables - des constantes

Ce code sera évalué par l'analyseur lexicographique.

3.3.4. Description de l'analyseur lexicographique

L'analyseur lexicographique [fig. 6] est constitué de trois sous modules [fig. 4] : - LEXDRI
- PROLUS
- INOUT

3.3.4.1. L'analyseur lexicographique est activé par l'analyseur syntaxique.

Ce dernier demande une nouvelle phrase Lucid [3.2] analysée lexicographiquement.

3.3.4.2. LEXDRI (lexical analyser driver)

LEXDRI identifie les différentes entités d'une phrase Lucid. Les entités étant toujours délimitées par des séparateurs, il est facile, au moyen de leur premier caractère, d'identifier la famille d'entité à laquelle ils appartiennent

EBCDIC	LUSCII	EBCDIC	LUSCII
0	0	Sp	37
1	1	+	38
2	2	-	39
3	3	*	40
4	4	/	41
5	5	(	42
6	6	)	43
7	7	,	44
8	8	.	45
9	9	¦	46
A	10	¢	47
B	11	!	48
C	12	;	49
D	13	:	50
E	14	?	51
F	15	|	52
G	16	¬	53
H	17	¬	54
I	18	=	55
J	19	@	56
K	20	¬	57
L	21	<	58
M	22	>	59
N	23	#	60
Ø	24	&	61
P	25	%	62
Q	26	ß	63
R	27		
S	28		
T	29		
U	30		
V	31		
W	32		
X	33		
Y	34		
Z	35		
_	36		

FIG. 5 : Caractères autorisés en Lucid

EBCDIC : caractère ebcdic
luscii : code correspondant en Luscii

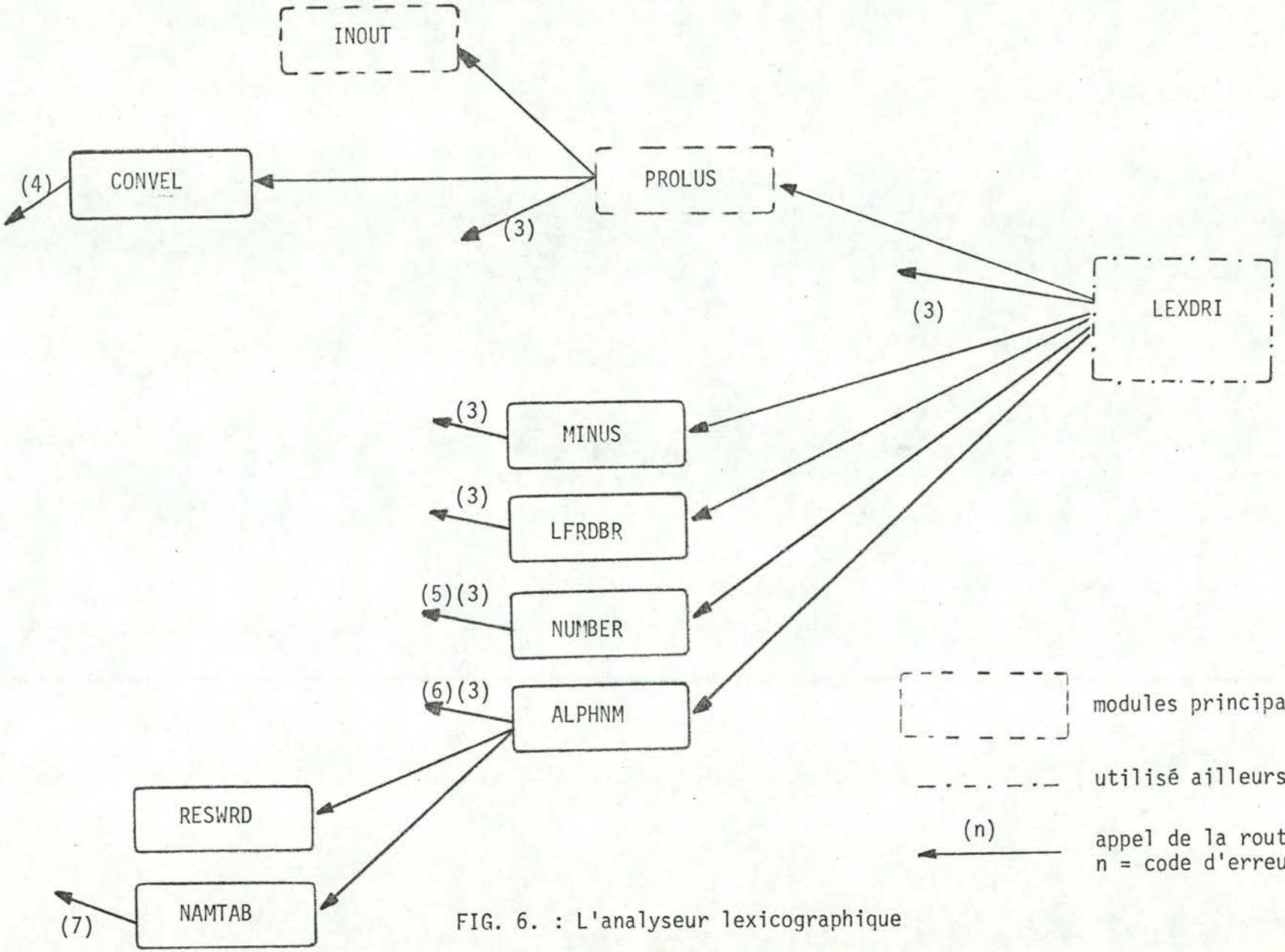


FIG. 6. : L'analyseur lexicographique

L'entité commence par :

- un chiffre : appel de la routine NUMBER
- une lettre : appel de la routine ALPHNM

Pour les séparateurs nous aurons :

- parenthèse ouvrante : appel de la routine LFRDBR
- l'opérateur "-" : appel de la routine MINUS
- blanc : n'est plus repris dans la représentation interne
- tout autre séparateur : représenté par son code Luscii

3.3.4.2.1. NUMBER

Reconstitue le nombre et teste s'il répond aux normes

3.3.4.2.2. ALPHNM

En Luscii [Fig. 5], le nombre maximum représentant une lettre ou un chiffre est 36. Il est donc aisé de reconstituer un code (nombre entier) représentant une chaîne de caractères commençant par une lettre : il suffit de procéder de la même façon que pour la reconstitution d'un nombre mais à raison de deux chiffres par caractère.

ex : ASA (as soon as)

sera représenté par :

((A(10) * 100) + S(28)) * 100 + A(10) = 102810

caractère

code Luscii de "A"

Contrainte matérielle

Pour des raisons de facilité d'implémentation sur le matériel Siemens, la longueur d'une chaîne de caractères ne peut excéder 4 : $3636363636 > 2^{31}$

Plus exactement, la longueur peut être quelconque mais seuls les 4 premiers caractères seront significatifs. ALPHNM cherche à représenter une chaîne de caractères au moyen d'un nombre. Deux types de symboles sont possibles :

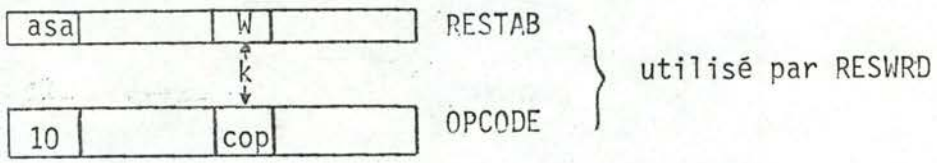
- mot réservé (la routine RESWRD (reserved word) teste si le symbole appartient à la table des mots réservés) auquel cas, le symbole est représenté par le code Lucode correspondant à ce mot réservé.

  ex : le nombre "10" représentera le mot réservé "asa" car le code opératoire de "asa" en Lucode est "10"

- Dans le cas contraire, la routine NAMTAB (name table) se chargera d'enregistrer, si ce n'est déjà fait, le nom de la variable dans sa table d'identification. Dans le programme Lucode, le symbole sera représenté par l'indice de sa position dans la table.

Tables utilisées

a) La table des mots réservés



Soit à voir si "W" est un mot réservé :

RESWRD teste si W appartient à la table RESTAB (recherche purement séquentielle)

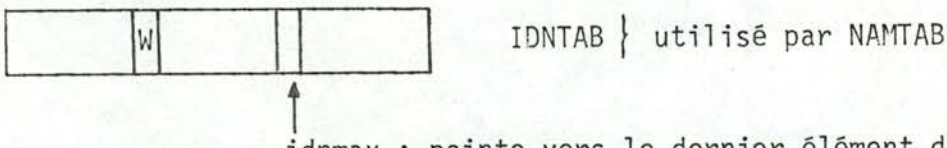
- si arrivé au bout de la table, "W" n'a pas été trouvé alors "W" n'est pas un mot réservé (1).
- sinon, $\exists$ k tel que RESTAB(k) = W
  W sera représenté par le code opératoire "cop" correspondant : OPCODE(k)

b) La table des identificateurs

Elle contient au départ les symboles :

- RES (result)
- OUT (output)

Ces variables seront ainsi représentées par un nombre fixé au départ. Ceci est nécessaire car il s'agit de la première variable à évaluer lorsque nous voulons évaluer une clause. Il faut donc connaître le nombre représentant ces variables.



"W" n'étant pas un mot réservé (1), nous chercherons s'il est dans IDNTAB

- la recherche jusqu'à idnmax est négative : nous l'ajoutons dans la table
  "W" sera représenté par IDNMAX + 1
- "W" est déjà dans la table IDNTAB : $\exists$ W/W = IDNTAB(j)
  "W" sera représenté par j.

3.3.4.2.3. MINUS

L'opérateur "-" (soustraction) unaire est transformé en un "-" binaire en utilisant zéro comme premier opérande.

ex : -b devient o-b

3.3.4.2.4. LFRDBR (left round bracket)

L'analyseur lexicographique fournit des codes distincts pour '(' selon son environnement :

- s'il s'agit d'une définition de clause
- s'il s'agit d'une référence à une clause

} ⇒ '(' ⇒ '56' (clause applicatio

- s'il s'agit d'une parenthèse dans une expression arithmétique '(' ⇒ '42'

ex : fun ny (Pagnol) devient fun ny 56(Pagnol)

Le séparateur '56' ne sert qu'à représenter les appels et les définitions de clauses. Il ne sera plus repris dans le programme Lucode.

3.3.4.2.5. Remarque

De la façon dont nous venons de représenter les différentes entités, un même nombre peut représenter plusieurs choses différentes.

exemples

- Le nombre "10" peut représenter :
  - . le nombre "10"
  - . le mot réservé "asa"
  - . la variable "x" tel que IDNTAB(10)=x
- Le nombre "40" peut représenter :
  - . le nombre "40"
  - . l'opérateur de multiplication "*"
  - . la variable "x" tel que IDNTAB(40)=x

Il faut cependant pouvoir différencier des entités de type différent. Nous avons pris pour convention :

. un opérateur sera représenté par le code Lucode correspondant, c'est-à-dire par un nombre ⩽ 63 [Fig.12]

(63 est le code Luscii maximal pour un caractère. Certains opérateurs (ceux réduits à un seul caractère : +, * ...) ayant leur code Lucode égal à celui en Luscii, "63" sera également le code Lucode maximal représentant un opérateur).

ex : * → 40

asa → 10

. une variable sera représentée par un nombre > 63

ex : si x = IDNTAB(k)

x sera représenté par : k + 63

. un nombre sera représenté par un nombre ⩾ 1000 (limite purement arbitraire)

ex : 719 sera représenté par 1719

3.3.4.3. PROLUS (produce Luscii)

- demande à INOUT [3.3.4.4.] de lire la première ou la prochaine carte du programme source
- demande à CONVEL [3.3.4.3.1.] de traduire ce qui vient d'être lu.
- s'il y a une carte de continuation ("c" en colonne 73), il répète le processus jusqu'à ce qu'il n'y ait plus de cartes de continuation. Nous obtenons ainsi une phrase Lucid [3.2.]

3.3.4.3.1. CONVEL (convert ebcdic to luscii)

Traduit chaque caractère en Luscii au moyen de la table de la fig. 5 (ex : 'c' devient 12)

Note : un message d'erreur est émis pour les caractères n'appartenant pas à la table

3.3.4.4. INOUT (input-output)

La routine lit une carte du programme source puis l'imprime.

3.4. L'analyseur syntaxique

L'analyseur syntaxique, activé par le rebaptiseur [fig. 4], fournit une première forme du Lucode. La deuxième forme, ou programme Lucode, s'obtient en rebaptisant les variables [3.5.]. L'apport de clauses au basic Lucid nous amène à parler de :

- variables globales
- paramètres formels - paramètres actuels
- appels de clauses

L'analyseur syntaxique devra donc traiter ces différentes particularités.

Pour faciliter l'exposé, nous allons scinder l'analyseur [fig. 8] en trois sous modules [fig. 4] traitant chacun un problème particulier :

- OPDRI : résoud le problème des variables globales
- SYNAN : résoud le problème des paramètres et produit les équations "Lucode"
- BETA : résoud le problème des appels de clauses.

3.4.1. Les variables globales

Considérons un programme ayant la structure suivante :

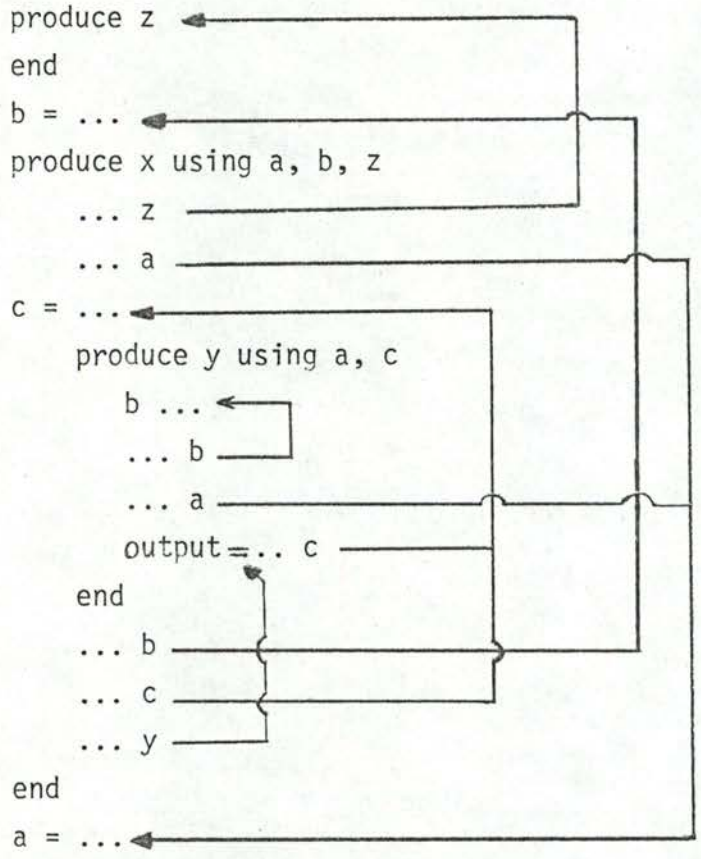


Fig. 7

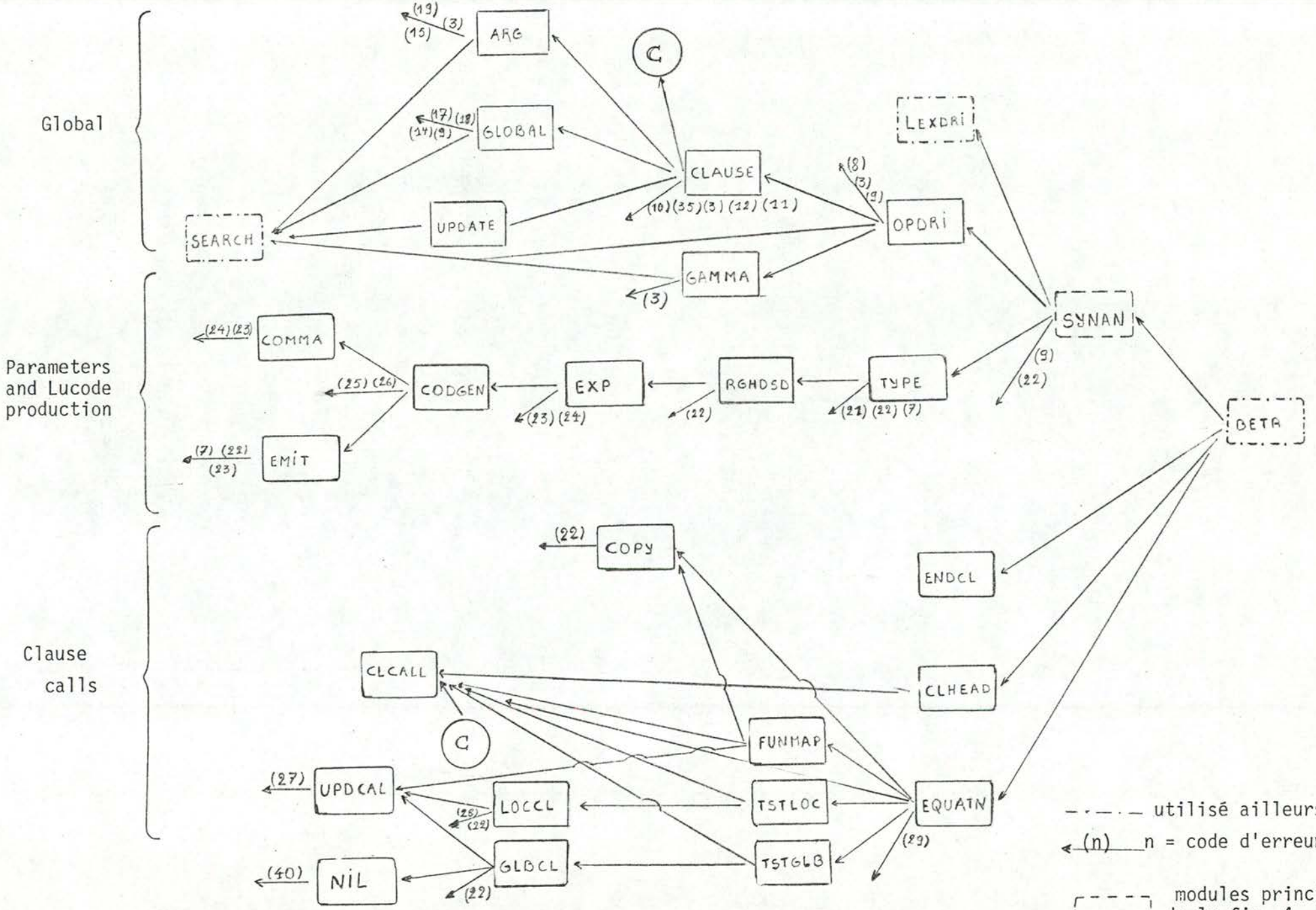


FIG. 8. Analyseur syntaxique

Les listes de variables globales permettent de voir, pour une clause donnée, la profondeur de définition de ces variables. Par profondeur de définition nous entendons un nombre de niveaux. Pour toute occurrence d'une variable globale, il faudra remonter ce nombre de niveaux pour retrouver la bonne définition de ces variables.

ex : [fig. 7]

. Dans la clause "x", il faut remonter à la clause de niveau supérieure pour retrouver la définition de "a" et de "b"

. Dans la clause "y", il faudra pour

a : remonter de deux niveaux

c : remonter d'un niveau

b : n'est pas dans la liste des variables globales; "b" a donc été définie dans la clause "y" même. Il s'agit d'une variable locale.

La première phase de l'analyse syntaxique consistera à :

- détecter les occurrences de variables globales
- établir le lien entre une occurrence d'une variable globale et la définition de cette dernière (clause où cette variable est considérée comme locale)

  Ce lien est représenté par un "pointeur" dans la fig. 7.

Dans une clause quelconque, une variable globale peut être

- une variable ordinaire ex : "a" [fig. 7]
- le nom d'une autre clause ex : "z" [fig. 7]
- le nom de la clause elle-même ex : function fac(n) using fac

  Ce dernier cas implique la récursivité

Remarque :

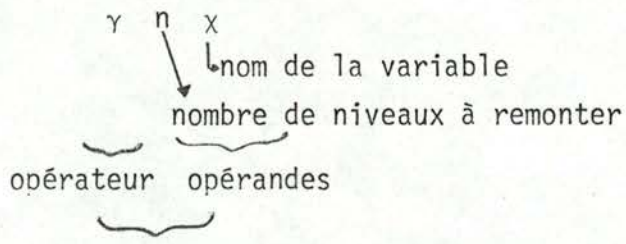
Par convention, le niveau le plus extérieur est appelé niveau 0. Par conséquent, descendre de n niveaux revient à incrémenter de n le nombre de niveau ("descendre" car la structure d'imbrication donne une structure en escalier).

"Remonter d'un niveau" est synonyme de : "revenir au bloc de niveau supérieur c'est-à-dire, celui contenant le bloc dont on fait l'analyse".

La détection d'occurrences de variables globales sera chose aisée : il suffit de tester, pour toute variable, son appartenance à la liste des variables globales. Dans le cas positif, il ne restera plus qu'à établir la bonne "connexion".

Comment procéder ?

- Pour toute occurrence d'une variable globale, nous introduisons un opérateur binaire de la forme :



représenté par un "pointeur" dans la fig. 7

L'opérateur "γ" permet, pour toute occurrence d'une variable globale, de retrouver le bon environnement (*) dans lequel elle a été définie.

Une fois les connexions établies pour toute une clause, nous pouvons supprimer la liste des variables globales de l'en-tête de la clause (Ceci sera démontré [§4]).

L'opérateur nous permet donc de dire que :

la valeur de "x" à cet endroit est la valeur que prend "x" dans la clause définie n niveaux plus haut.

Nous aurions pu résoudre le problème des globaux d'une façon toute différente. Il est en effet possible d'éliminer complètement la structure de bloc dès l'analyse lexicographique, et par conséquent le concept même de variable globale.

En effet, la structure de bloc a pour but de permettre l'emploi d'un même identificateur pour désigner, dans des blocs différents, des objets différents. L'introduction de la notion de variable globale permet alors aux identificateurs désignés de garder la même sémantique au travers d'une frontière de bloc. L'analyseur ayant entre autre pour fonction d'associer à chaque identificateur du programme source un numéro qui est sa traduction en Lucode, il suffirait donc que deux variables de signification différente (définition différente) aient

(*) L'environnement à un moment donné est l'ensemble des variables et des valeurs y associées accessibles à ce moment.

des noms en Lucode différents, c'est-à-dire soient représentées par des nombres différents.

Soit un programme du type :

```
A = ...
B = ...
C = ...
D = ...
produce X using A, B
    C = A + D
    D = B
    output = C + D
end
```

La première méthode donne :

```
A = ...
B = ...
C = ...
D = ...
produce x
    C = γ1 A + D
    D = γ1 B
    output = C + D
end
```

tandis que par la deuxième méthode nous aurions obtenu :

```
A = ...
B = ...
C = ...
D = ...
C* = A + D*
D* = B
X = C* + D*
output X
```

où C* est la représentation de la variable locale C

C est la représentation de la variable globale C

Nous avons choisi la première méthode pour plusieurs raisons.

- Cette deuxième méthode semble présenter un avantage important : la relation entre les identificateurs Lucode et les objets qu'ils désignent est biunivoque dès l'analyse. L'interpréteur est ainsi libéré de la tâche consistant à resituer un identificateur dans le contexte approprié (on pourrait dire que l'opérateur "γ" est compilé plutôt qu'interprété).

Mais cette façon de faire complique au contraire singulièrement la tâche de l'interpréteur. En effet, dans la première méthode, le corps de chaque clause reste parfaitement distinct. Lors de l'appel d'une clause, l'interpréteur peut donc générer un nouvel environnement propre à cette clause. Au contraire, en appliquant la deuxième méthode, l'interpréteur ne peut plus distinguer à priori au début de l'évaluation d'une clause l'environnement nécessaire propre à cette dernière. Deux solutions sont alors possibles : ou bien on considère un environnement certainement suffisant, c'est-à-dire l'ensemble de toutes les variable connues, ou bien on évalue cet environnement au fur et à mesure des besoins. Dans les deux cas, le travail de l'évaluateur est bien plus considérable que s'il avait à prendre en charge l'opérateur "γ".

Cada sueno es anoncio de una verdad, mejor de una realidad, que existe cerca de nosotros, sin que tengamos consciencia de ella.

ou mieux encore :

Tout rêve est le signe d'une vérité,
mieux : d'une réalité
présente auprès de nous sans que nous en ayons conscience

(G. MARTINEZ - SERRA)

- Il n'est pas impossible d'écrire un interpréteur jouissant des deux propriétés suivantes :

  - Lors de l'évaluation d'une clause, il n'est pas nécessaire de garder le Lucode correspondant à tout le programme en mémoire centrale : celui de la clause correspondante suffit.
  - L'interpréteur, basé sur le principe du "call by need" pourrait évaluer l'output de certaines clauses en parallèlle.

Dans les deux cas, il faudrait pouvoir accéder au corps de chacune des clause La structure de bloc ne pourrait donc disparaître.

Nous allons décrire maintenant les routines traitant le cas des variables globales [fig. 8]

3.4.1.1. OPDRI (gamma operator driver)

La routine sert d'aiguillage aux différentes phrases reçues de l'analyseur lexicographique.

Si la phrase est :

- l'en-tête d'une clause : elle est analysée par le routine CLAUSE [3.4.1.3.]
- la fin d'une clause : mise à jour de pointeurs
- une équation : elle est analysée par la routine GAMMA [3.4.1.2.]

3.4.1.2. GAMMA

La routine introduit l'opérateur "γ" pour les variables globales. Le "nombre de niveaux" à remonter (premier opérande de "γ") est disponible dans une table mise à jour, lors de chaque définition de clause, par la routine UPDATE [3.4.1.6. Dans le cas de clauses de type "mapping" ou "compute", il faut introduire, pour les variables globales, non seulement l'opérateur "γ" mais aussi l'opérateur "λ" (latest [1]). Ces deux types de clauses ont en effet la particularité de "geler" le temps des variables globales [chap. I]

Considérons l'exemple :

```
produce X using A, B
    C = ... A ... B
    produce Y using A, C
        B = ...
        ... A ... C                    P
        compute Z using A, C, B
            ... A ... C ... B
            mapping W(i) using A, B
                ... A ... B
            end
        end
    end
end
```

Les clauses "map" et "cmp" pouvant être considérée comme un combiné de "latest" avec respectivement "fun" et "pro", la première étape consistera à introduire "λ"

Le programme P devient :

```
pro X usg A, B
    C = ... A ... B
    pro Y usg A, C
       B = ...
       ... A ... C
       pro Z usg A, C, B
         ... λ1A ...λ1C ...λ1B
         fun W(i) usg A, B
             ... λ(λA)... λ(λB)
         end
       end
    end
end
```

P'

Plutôt que d'écrire $\lambda(\lambda(\lambda \ldots (\lambda A))) = \lambda^n A$ nous noterons λnA

$$\lambda nA|_{t_o \ldots t_k} = A|_{t_o \ldots t_k t_{k+1} \ldots t_{k+n}}$$

Appliquer n fois l'opérateur latest sur une variable revient à accroître de n le nombre de paramètres temps [1]

La deuxième étape consiste à introduire "γ"

```
pro x
   C = ...  γ1A ...  γ1B
   pro Y
     B = ...
     ...  γ2A ...   γ1C
     pro Z
       ... λ1(γ3A)    λ1(γ2C)    λ1(γ1B)
       fun W(i)
           ... λ2(γ4A) ... λ2(γ2B)
       end
     end
   end
end
```

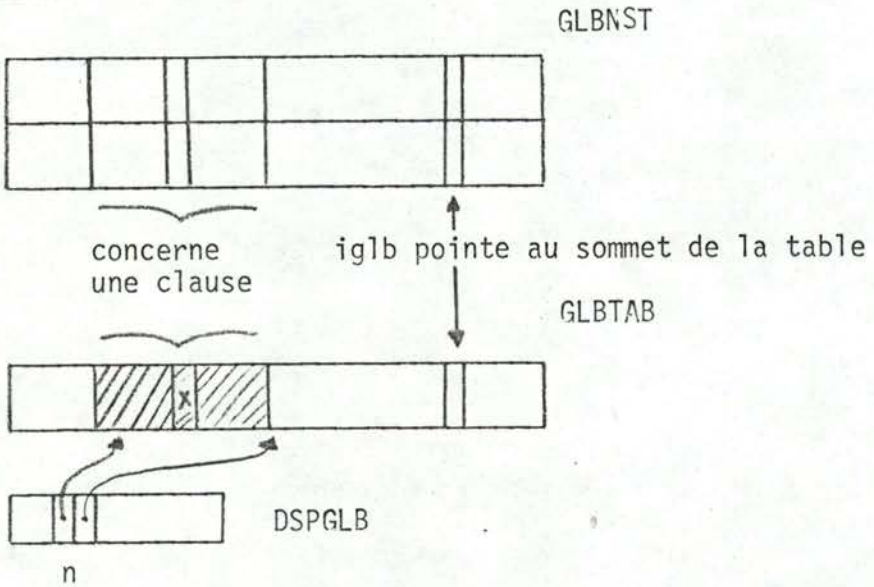
"λ" et "γ" sont tous deux des opérateurs binaires. Le premier opérande de "λ" n'étant pas toujours égal à celui de "γ", il faudra garder trace de deux choses

- nombre de niveaux à remonter
- nombre de fois qu'il faut appliquer "latest".

Ces deux opérateurs jouent des rôles forts similaires : tous deux permettent de retrouver le bon contexte de définition d'une variable

- "λ" du point de vue temps
- "γ" du point de vue environnement dans la mesure où chaque clause définit un nouvel environnement.

Tables utilisées



Soient . n le numéro de clause en cours d'analyse
. x la variable rencontrée

La liste des variables globales d'une clause est stockée séquentiellement dans une table : GLBTAB (global table). L'adresse du début de la liste dans GLBTAB est mémorisée dans une autre table DSPGLB (display for global).
Si "X" est une variable globale, alors :

$$X \in [GLBTAB(DSPGLB(n), GLBTAB(j)]$$

(= partie hachurée)

$$\text{où } j = \begin{cases} iglb & \text{si n est le bloc de "plus grand poids" rencontré jusqu'à présent} \\ DSPGLB(n+1) & \text{sinon} \end{cases}$$

BLOC DE PLUS GRAND POIDS

Considérons un programme ayant la structure suivante :

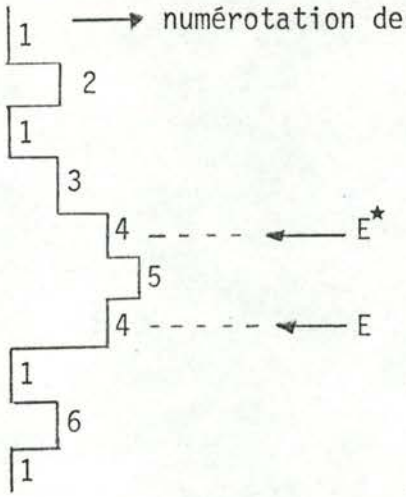


FIG. 9

Supposons que nous analysions l'équation E^*. Nous définissons, pour E^*, le bloc de plus grand poids comme étant le bloc n° 4.
Pour E, ce sera le bloc n° 5

soit i tel que GLBTAB(i) = X

Le nombre de niveau à remonter (premier opérande de "γ") sera : GLBNST(1, i)

Le nombre de fois à remonter dans le temps (premier opérande de "λ") sera GLBNST(2, i)

Note : - il n'y a pas lieu d'introduire "λ" si GLBNST(2, i) = 0

3.4.1.3. CLAUSE (clause définition)

- Dans le cas de clauses à arguments, appel de la routine ARG [3.4.1.4.] qui analysera la liste des paramètres formels.
- Si la clause a des variables globales, appel de la routine GLOBAL [3.4.1.5] qui analysera cette liste.
- Construction d'une arborescence où un sommet est le nom d'une clause.
  ex : supposons que le numéro de bloc corresponde au nom de la clause dans la fig. 9. L'arborescence correspondant à cette figure sera :

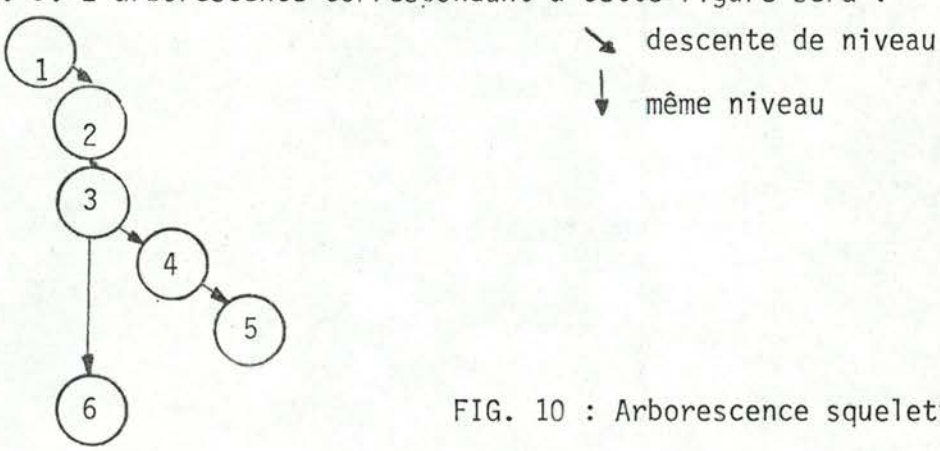


FIG. 10 : Arborescence squelette

Construction de l'arborescence

Soient "n" le nom de la clause en cours d'analyse

"m" le nom d'une clause définie dans "n" (descente de niveau)

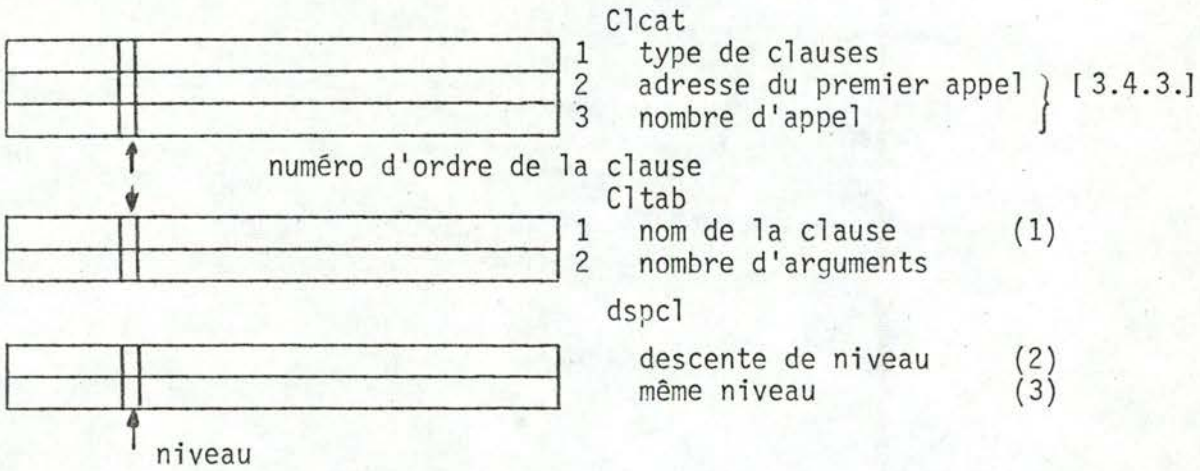
- n possède déjà un sous arbre de droite
  → ajouter un sommet "m" à ce sous arbre en suivant les sous arbres de gauche de ce dernier
- sinon : ajouter au sommet "n" le sous arbre de droite dont la racine et unique sommet est "m".

Caractéristiques de cette arborescence

- Pour une clause donnée, nous pouvons
  - descendre de niveau
    { et / ou } lors de la définition d'une nouvelle clause
  - rester au même niveau (dans ce cas, il y a chaînage)

  Nous obtenons donc une arborescence binaire.
- De la racine ne peut partir qu'au plus un arc (descente de niveau)

Tables utilisées



(1), (2) et (3) constituent l'arbre squelette

(1) = sommet

(2) (3) type d'arc

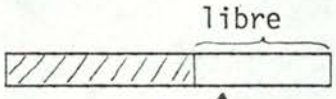
Note : Le numéro d'ordre d'une clause est l'ordre d'apparition de sa définition [fig. 9]

3.4.1.4. ARG (arguments)

La routine mémorise les paramètres formels dans une table. Cette dernière sera utilisée ultérieurement lors de la production des équations Lucodes correspondant à ces paramètres.

Note : Un diagnostic d'erreur est émis lors d'un double emploi d'un même argument ex : fun(i, i)

Table utilisée



iarg pointe vers le premier élément libre dans PARTAB
PARTAB sera de nouveau entièrement disponible après la production du Lucode c'est-à-dire pour la prochaine définition de clause.

3.4.1.5. GLOBAL

Cette routine joue le même rôle que ARG mais pour les variables globales. Celles-ci sont supprimées de l'en-tête de la clause et sont mémorisées dans GLBTAB [3.4.1.2.] jusqu'à la fin de la phase de "transformation".
(Lors de la phase de rebaptisation, il faudra pouvoir accéder aux variables globales de chacune des clauses)
GLOBAL détecte aussi :

- double emploi de variables globales
- l'emploi d'une variable globale en tant que paramètre formel.

3.4.1.6. UPDATE

La routine "met à jour" la table GLBNST [3.4.1.2.]

Le nombre de niveaux à remonter, est incrémenté de 1 pour les variables globales de clauses imbriquées.
Le nombre de fois qu'il faut appliquer latest sera incrémenté de la même façon dans le cas de clauses "cmp-map".

La "mise à jour" se fait donc en fonction de la liste des variables globales de la clause de niveau supérieur.

3.4.1.7. SEARCH

Il s'agit d'une fonction standard qui, pour un élément name, cherche séquentiellement dans une table unidimensionnelle ("tab") l'indice i tel que :
tab(i) = name

$$\text{SEARCH} = \begin{cases} i \\ o \text{ si name} \notin \text{tab} \end{cases}$$

3.4.2. Les paramètres formels et la production du Lucode (1ère forme)

Nous n'avons, jusqu'à présent, supprimés de l'en-tête d'une clause que les variables globales. La deuxième phase de l'analyse syntaxique consistera entre autre à supprimer les paramètres des en-têtes.

Mise au point

- Nous aurions pu supprimer les paramètres, de l'en-tête d'une clause, lors de la première phase de l'analyse syntaxique, étant donné qu'ils sont mémorisés dans une table. Nous avons cependant préféré attendre de telle sorte qu'un problème (ici le problème des paramètres) soit résolu entièrement au cours d'une phase. Ceci a l'avantage d'avoir en output d'une phase de transformation un programme et non un programme qui n'a de sens qu'au travers de tables d'informations.
- Les clauses "map-fun" sans paramètres sont équivalentes à des clauses "pro-cmp". Afin de prévoir l'oubli, de la part du programmeur, de la spécification des paramètres, un message d'erreur sera émis pour les clauses "fun-map" sans arguments.
- Pourquoi "supprimer" la liste des paramètres formels ?
  L'interpréteur ne travaille que sur des équations. Les en-têtes vont donc devoir disparaître. Il faudra traduire la notion de paramètre formel au moyen d'une équation.

Une première idée consiste à, partant de l'en-tête $f(A_1 \ldots A_n)$, remplacer toute occurrence d'un A_i dans le corps de la clause par αi

où α = opérateur unaire signalant qu'il s'agit d'un paramètre formel
i = numéro d'ordre du paramètre

Cette solution n'est cependant pas optimale : elle ne résout pas complètement le problème. Considérons, à cet effet, l'exemple suivant :

```
fun f(A)                                  fun f(A)
  ... A                                     ... A
  pro X usg A                               pro X
  V = A          ---1ère phase de-->        V = γ1 A
  end                                       end
end          l'analyse syntaxique         end
                                          end
```

Fig. 11

Si on remplace A par α_1, il ne sera plus possible d'évaluer V car il n'existe pas d'équations correspondant à A.

Plutôt que de substituer α_1 à A, nous produirons, avant d'analyser le corps de la clause, l'équation $A = \alpha_1$

Toute occurrence de "A" dans le corps de la clause se référera automatiquement à cette équation.

Généralisation

Pour $f(A_1, A_2, \ldots A_n)$ nous produirons au début du corps de la clause, les équations "$A_i = \alpha i$"

Dans le cas de clauses "map", il faut également introduire l'opérateur "λ" pour les paramètres formels. Cet opérateur sera introduit en même temps que "α".

```
ex : map OM(C, MOI)                    pro OM
        ... C                             c = λ1 T1
        ... MOI          devient          T1 = α1
     end                                  MOI = λ1 T2
                                          T2 = α2
                                          .... C
                                          .... MOI
                                       end
```

où T_i = variables temporaires.

Note : En introduisant . "λ" : map est ramené à fun
. "α" : fun est ramené à pro

Une autre solution, proposée par Mr Wadge, fut d'employer "α" en tant qu'opérateur binaire, pour toute occurrence d'un paramètre.

Dans l'exemple ci-dessus, au lieu d'avoir

γ_1 A : nous aurions eu $\alpha_{2\,1}$

α_1 $\alpha_{1\,1}$

numéro du paramètre

nombre de niveaux à remonter

Nous n'avons pas adopté cette solution pour deux raisons :

- Si un paramètre apparaît plusieurs fois dans le corps de la clause, il y aura, pour chacune de ces occurrences, la production d'une équation :

  "$T_j = \alpha nm$"

  où T_j est une variable temporaire et différente pour chacune de ces équations.

Nous aurions plusieurs de ces équations alors qu'une seule aurait suffit.

- L'analyse d'une clause doit être aussi indépendante que possible du reste du programme. (l'en-tête étant un "interface" suffisant). Ceci n'est plus respecté avec la dernière méthode. Il faut en effet pour une variable globale tester si celle-ci est un paramètre formel dans la clause de niveau supérieur, auquel cas il faudrait utiliser "α" plutôt que "γ".

L'autre fonction de cette phase de l'analyseur syntaxique est de produire le Lucode première forme. Le Lucode produit ici contient encore des phrases telles que : - en-tête (raccourcie) de clause

- "end" (fin de clause), alors que l'interpréteur n'accepte que des équations.

Les équations Lucodes sont obtenues en faisant "exploser" une équation contenant plusieurs opérateurs, en plusieurs équations n'ayant chacunes qu'un et un seul opérateur. Il s'agit d'un processus classique basé sur la priorité et la parité des opérateurs.

ex : x = a + b * c / d devient

$T_1 = c/d$

$T_2 = b*T_1$

$x = a + T_2$

3.4.2.1. SYNAM (syntaxical analyser)

Jusqu'à présent, l'analyse s'est faite "phrase par phrase". Le programme doit avoir été complètement lu pour la troisième phase de l'analyseur syntaxique. (celle-ci aura en effet besoin de l'arbre squelette [fig. 10] complètement construit).
SYNAM va donc reconstituer le programme sous sa nouvelle forme.

3.4.2.2. TYPDRI (type driver)

Guide le traitement à faire selon les types de phrases fournies à cette phase :

- équation [3.4.2.3.]
- fin de clause
- en-tête d'une clause

  Si clause à arguments - production des équations "A_i [=] αi"
  - introduction, si nécessaire, de "λ"
  - suppression de la liste des paramètres dans l'en-tê

3.4.2.3. RGHDSD (right hand side)

- produit directement l'équation Lucode pour les équations sans opérateur en utilisant le signe d'égalité comme opérateur.

  ex : x = y devient x [=] = y

- dans le cas contraire, l'analyse sera faite par EXP [3.4.2.4.]

3.4.2.4. EXP (expression)

- analyse une expression en utilisant deux stacks
  - . un pour les opérateurs
  - . un pour les opérandes

  Rappellons que les opérateurs sont facilement reconnaissables : ils sont représentés par un nombre $\leqslant 63$.

- Un opérateur est ajouté au sommet de la pile s'il est de priorité plus forte que l'opérateur au sommet.

  La priorité des opérateurs est connue au moyen de :
  - . priorité de droite à gauche
  - . règles de priorités données dans une table

  [Fig. 12 : PRECEDENCE]

- Dans le cas contraire, une équation Lucode sera produite [3.4.2.5.]

  La parité des opérateurs est connue au moyen d'un tableau [Fig. 12 : ARITY]
  Un opérateur de parité négative ne peut apparaître dans le texte d'une expression.

- Les variables sont toujours ajoutées au sommet de la pile et n'en sont retirées que lors de la génération d'une nouvelle équation.

3.4.2.5. CODGEN (code generator)

Dans le cas d'un n-uple, appel de la routine COMMA [3.4.2.7.], la routine fournit ensuit à EMIT [3.4.2.6.] ce dont elle aura besoin pour produire l'équation :

- . opérateur
- . nombre adéquat d'opérandes

L'analyse de l'expression ne se terminera que lorsque la pile des opérateurs est vide.

3.4.2.6. EMIT (emit Lucode)

produit l'équation en introduisant une variable temporaire; variable temporaire qu'il faudra remettre au sommet de la pile des variables.

LUCODE	Arity	Preced nce	Opérateur-séparateur	mot réservé
0	-1	0		
1	1	99	alpha	
2	-1	0	beta	
3	2	98	gamma	
4	2	97	latest	
5	-1	0		
6	-1	0		
7	-1	0		
8	-1	0		
9	-1	0		
10	2	5	as soon as	asa
11	-1	3	begin of expression	
12	-1	0	compute	cmp(*)
13	-1	0	end	end
14	-1	0		
15	1	95	first	frst(*)
16	-1	0	function	fun(*)
17	-1	0		
18	-1	3	if	if
19	-1	0		
20	-1	0		
21	-1	0		
22	-1	0	mapping	map(*)
23	1	95	next	nxt(*)
24	-1	0	using	usg(*)
25	-1	0	produce	pro(*)
26	2	97	cons	
27	1	25	not	not
28	-1	0	false	fals(*)
29	-1	0	true	true
30	2	5	upon	upon
31	-1	0		
32	2	5	whenever	when(*)
33	2	40	or	or
34	2	5	followed by	fby
35	2	30	eq	eq
36	-1	0		
37	-1	0		
38	2	60	+	
39	2	60	-	
40	2	70	*	
41	2	70	/	
42	0	3	)	
43	0	3	(	
44	-1	0		
45	-1	0		
46	-1	0		
47	-1	0		
48	-1	0		
49	2	20	else	else
50	2	10	conditional	then
51	-1	0		
52	-1	0	inv charact	
53	0	1	none end of line	
54	-1	0	none end of file	
55	-1	0	identity	
56	2	98	clause application	
57	-1	0		
58	2	30	<	

LUCODE	Arity	Precedince	Operateur séparateur	mot réservé
59	2	30	>	
60	2	80	e	exp
61	2	40	And	And
62	2	80	**	pow(*)
63	2	80	ln	ln

Note : (*) le mot complet peut être utilisé (ex : fun ou function
pow ou power)

FIG. 15.

3.4.2.7. COMMA

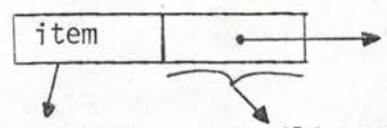
Rappellons qu'une équation Lucode est constituée :

. d'un opérateur

. de deux opérateurs maximum.

Il faudra donc mettre un n-uple sous cette forme.

Partant d'un n-uple de variables, (dans notre cas n-uple de paramètres actuels) la routine COMMA crée une liste : chaînage entre les différentes composantes. Un élément de la chaîne sera une paire pointée



une composante de n-uple

élément chaîneur : pointe vers la paire pointée suivante.

Dans le n-uple, deux composantes sont séparées au moyen d'une virgule. Le chaînage se fait au moyen de l'opérateur "cons".

Le chaînage s'obtient en - transmuant toutes les virgules en "cons".
- insérant une variable temporaire (élément chaîneur) entre deux composantes (fait automatiquement par CODGEN et EMIT)

dans le n-uple.

La dernière paire pointée aura pour seconde opérande l'atome "nil".

Exemple : X = f(A, B, C)

devient

sera définie en [3.4.3.]

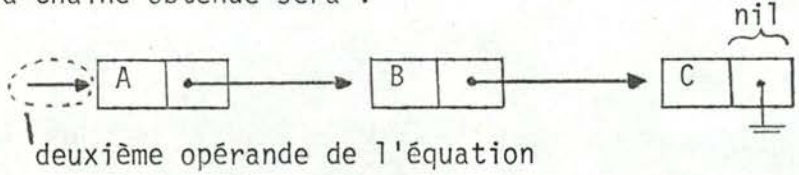
X = ... f T_3

T_1 = cons c nil

T_2 = cons B T_1

T_3 = cons A T_2 où les Ti sont des variables temporaires

La chaîne obtenue sera :



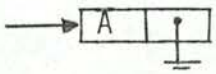
deuxième opérande de l'équation traduisant un appel de clause (T_3 dans notre cas)

Cas particuliers

. n-uple pour n=1

ex : f(A)

Il n'y a pas de problème



. n-uple pour n = 0

ex : X = P où P = nom d'une clause "pro ou cmp"

Là non plus, il n'y a pas de problème : "→" devient ⊥

Le deuxième opérande de l'appel de clause sera l'atome nil.

Remarque :

Nil étant une constante, il aura une représentation analogue à celle-ci. Cependant pour éviter toute confusion avec la représentation des nombres, nous allons représenter nil par 700 ($\leqslant$ 1000)

Ordinogramme de l'analyse d'une expression

. stack des opérandes = VRST indicé par iVR

. stack des opérateurs = OPST indicé par iOP

. P = priorité

. A = parité

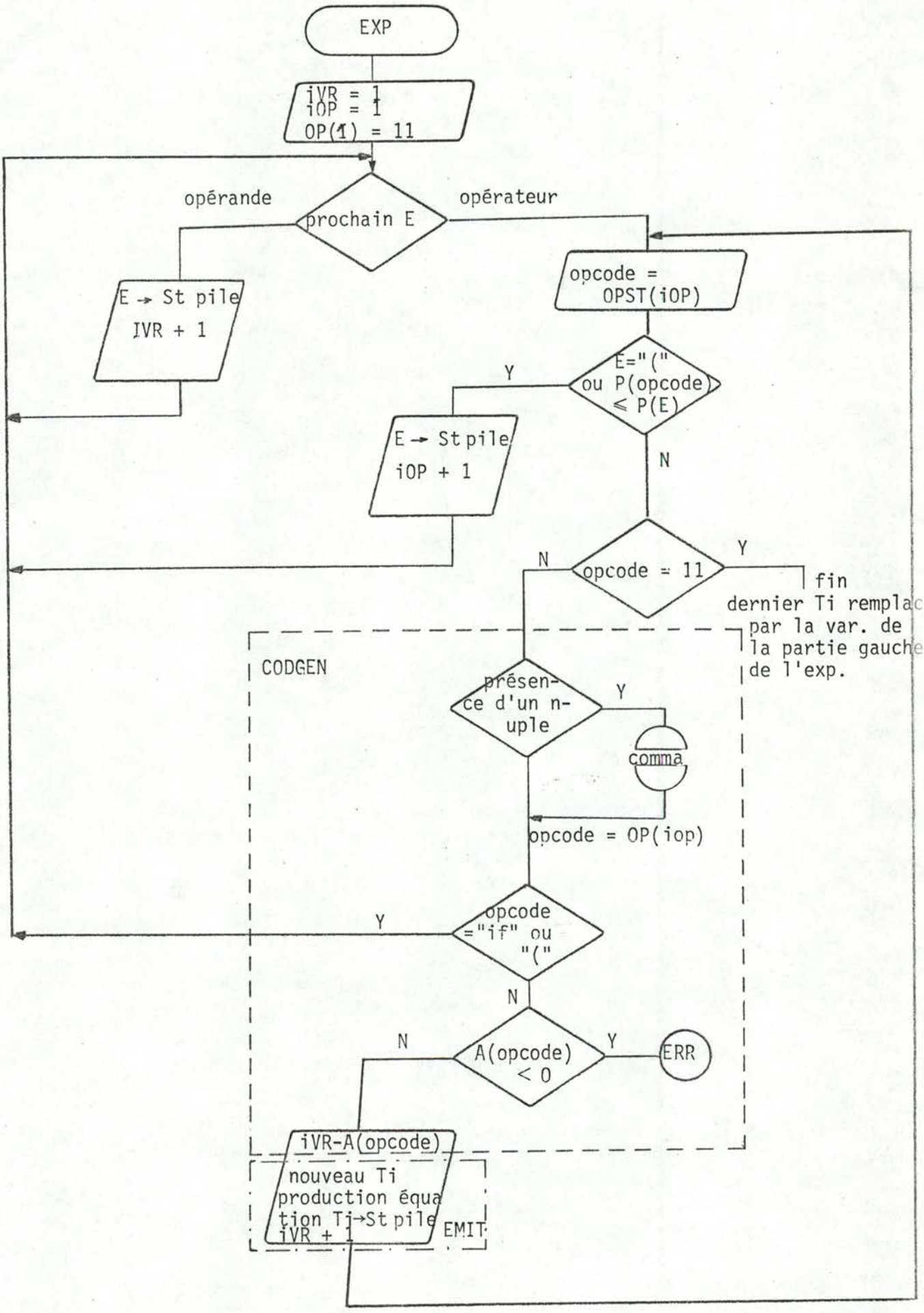
. E = élément de l'expression

. 11 = début d'expression

. Ti = variable temporaire

. Var = variable

. exp = expression



N.B. : St = Sommet d'une pile

3.4.3. Les appels de clauses

Il s'agit, avec les variables globales, du problème le plus délicat. Considérons le morceau de programme suivant :

```
X = f(U, V, W)  (1)
fun f(A, B, C)
  ¦
  out = ...
end
```

De par la définition d'un appel de clause, il faudrait pouvoir écrire :

```
X = out f où A = U
            B = V
            C = W
```

La variable "out" est identifiée par le même nombre, quelle que soit la clause dont elle est la variable output. Il faut trouver un moyen de lier "X" à la variable "out" de la clause de nom "f".

Partant de l'équation (1), nous aurions tendance à écrire :

```
X = f          (2)
f = ... n L    (3)
    ¦ ¦ ¦
    ¦ ¦ ¦ lien avec les paramètres actuels
    ¦ ¦ numéro de clause
    ¦ opérateur à spécifier
```

- Mais l'équation "X = f" n'a aucune signification mathématique : "f" a lui tout seul n'ayant aucune signification.
- De plus, pourquoi vouloir parler d'un numéro de clause ? Ayant gardé la structure d'imbrication, il faudra à un moment donné introduire un "display" permettant de retrouver le début de chacune de ces clauses.
  Un tableau (le display) étant indicé par un nombre, il paraît normal de parler d'un numéro de clause.
  Mais n'est-ce pas là une fausse interprétation de l'équation (1) ? Nous ne voulons pas mettre en rapport "X" et l'output de la clause n° n mais bien "X" et l'output de la clause de nom "f".

Partant de ces deux remarques, nous allons condenser les deux équations (2) et (3) en une seule.

```
X = ... f L
      ↘ opérateur à spécifier
```

L'opérateur devrait permettre de dire : "la valeur de X est la valeur que prend la clause f étant donné que les paramètres actuels sont donnés par la liste dont "l'adresse" de début est L.

La fonction de l'opérateur étant définie, nous le baptisons "BETA" (β).

L'équation (1) pourra enfin s'écrire :

$X = \beta f^{\star} L$

T_1 = cons C nil

T_2 = cons B T_1

L = cons A T_2 où T_i = variable temporaire

Signification du $f^{\star}$

Jusqu'à présent, f était représentée de la même façon qu'une variable. Comme le nom d'une clause est une constante, il faudra donner à f une nouvelle représentation.

Nous allons élargir le champs des constantes, jusqu'à présent représentées pa un nombre $\geqslant$ 1000 (sauf nil représenté par 700). Le nom d'une clause sera représenté par un nombre $\geqslant$ 500.

(variable :]63, 500[ )

Nous allons nous arranger pour que la représentation du nom d'une clause soit, également, porteur d'une double information :

- il s'agit d'un nom de clause
- il spécifie de quel numéro de clause il s'agit !

Il existe bien une relation biunivoque entre le nom d'une clause et sa représentation.

L'interpréteur distinguera donc :

- opérateurs : nombre $\leqslant$ 63
- opérandes : nombre > 63 . nombre $\geqslant$ 1000 nombre
  . nombre = 700 nil
  . 500 $\leqslant$ nombre < 700 nom de clauses
  . 63 < nombre < 500 variables

Remarques

- Le nom d'une clause à arguments ne peut apparaître que dans des équations en
  - β
  - γ
  - cons

exemples :

. X = + A f^*

variable — nom d'une clause

Nous venons de voir que l'équation n'a pas de sens

. X = γ n f^*

X est le nom de la clause définie n niveaux plus haut et appellée "f^*"

Note : nous confondrons, pour facilité d'écriture, le nom d'une clause et sa représentation.

X est donc synonyme de f^*

Il n'y aura donc pas de confusion en écrivant :

"A = β X L" car une fois "évalué", X sera bien le nom de la clause à activer.

. X = cons f^* T_1

Le nom de la clause à activer est le nom passé, en tant que paramètre, à la clause contenant cette équation.

- Rappelons que les appels de clauses à arguments ont déjà été identifiés par l'analyseur lexicographique.
  Celui-ci avait introduit un séparateur spécial.
  Pourquoi ne pas avoir utilisé directement "β" ? "β" est un opérateur traduisant un appel de clause alors que le séparateur servait aussi bien aux appels qu'aux définitions de clauses

Il reste à résoudre le problème des clauses "produce-compute". Une équation de la forme X = A+p (4), où p est le nom d'une clause, est légale et signifie : la variable X prendra les mêmes valeurs que la somme des valeurs de "A" et de "p"; "p" étant défini par un ensemble d'équations.

Il est donc possible de lier directement l'équation (4) avec l'output de la clause "p". Il suffit de :

- remplacer la variable output de cette clause par "p".
- rebaptiser les variables locales pour éviter les interférences entre définitions de variables.

```
ex : X = ... p
     pro p
       X = ...
       out = ...
     end
```

```
X = p ┐
  ⋮   │ non légal en Lucid
X = ...┘
  ⋮
p =
```

Ceci revient à supprimer la structure d'imbrication dans le cas de clauses sans argument. Nous avons déjà rejeté cette hypothèse quelle que soit le type de clause [3.4.1.]

Nous allons donc considérer les équations de type (4) comme étant un appel de clause tout comme dans le cas des clauses à arguments.

(4) → X = + A T
T = β p* nil

Remarques

- Les définitions de clauses sont toutes ramenées à des définitions de "produce" en introduisant :
  . l'opérateur "α" pour les paramètres formels des clauses "fun-map"
  . l'opérateur "λ" pour
    - les paramètres formels des clauses "map"
    - les variables globales des clauses "map-cmp"
- Les appels de clauses sont toutes ramenées, quant à elles, à des appels de clauses de type "function" ou "mapping" en :
  - introduisant l'opérateur "β" pour les clauses "pro-cmp"
  - remplaçant la "clause application", introduite par l'analyseur lexicographique, par l'opérateur "β".

- L'opérateur latest n'a de sens que si un événement a "gelé" le temps auparavan Cet événement surgit lors de l'appel d'une clause "cmp-map". L'interpréteur disposant d'un catalogue des clauses utilisées, gèlera le temps lorsque l'opérateur "β" s'applique à une clause du type cité plus haut. L'opérateur "β" joue un rôle différent selon le type de clause, c'est pourquoi nous avons distingué "appel de type fun" ou "de type map".

La deuxième phase de l'analyseur syntaxique joue également un autre rôle : el mémorise, pour l'interpréteur, des informations concernant les clauses. Nous verrons [§5] que l'interpréteur a besoin de trois informations concernant les clauses :

- type de clause : clause qui gèle le temps ou pas
- numéro d'ordre de l'appel de la clause
- nombre d'appels possibles à l'intérieur de la clause

La première information a été enregistrée lors de la définition de la clause. Nous allons de nouveau considérer le numéro d'ordre comme étant l'ordre d'apparition de l'appel à l'intérieur d'une clause. Une clause peut être appelée de plusieurs endroits différents, d'où la question : comment l'interpréteur peut-il rentrer en possession du numéro d'appel ?

Une première méthode, primitive, consiste à avoir dans une table les différents endroits d'appel de cette clause. Lors de l'appel, un petit balayage permettra de dire duquel il s'agit.
Ce système est assez lourd et donc à éviter.
Une autre idée, qu'un programmeur aurait trop souvent tendance à utiliser, est de "glisser" le numéro d'appel dans la liste des paramètres actuels. (ex : le premier paramètre)
Mais ce paramètre n'est d'une part qu'une donnée nécessaire pour l'interpréteur et d'autre part, n'appartient pas au programme source Lucid. Nous ne pouvons donc transformer le programme suivant cette méthode. Nous allons plutôt profiter d'une caractéristique de la représentation des équations Lucodes.

Le programme Lucode final (celui qui sera interprété) sera mémorisé dans des vecteurs de telle sorte que les éléments de même indice reconstituent une équation.

Nous allons regrouper tous les appels entre eux de telle sorte que, connaissant

- l'adresse du premier appel (indice : l^*)
- l'adresse de l'appel actuel (indice : l),

l'interpréteur puisse retrouver le numéro d'appel, soit "$l-l^*+1$".

La phase de "nouvelle identification" des variables regroupera les appels entre eux. Quant à la troisième phase de l'analyseur syntaxique, elle se chargera, en plus d'introduire "β", de mémoriser dans une table

- type de clause
- adresse du premier appel
- nombres d'appels possibles

3.4.3.1. BETA

guide l'analyse à faire suivant le type de phrase reçue de la deuxième phase de l'analyseur syntaxique :

- en-tête d'une clause
- équation
- fin de clause : "end"
- fin de fichier

3.4.3.2. CLHEAD (clause head)

mise à jour de pointeurs

3.4.3.3. ENDCL (end of clause)

mise à jour de pointeurs

3.4.3.4. EQUATN (equation)

Différents traitements sont possibles suivant le type d'équation :

. l'équation est déjà en quelque sorte un appel de clause (elle contient une clause application) "β" remplacera cet opérateur et la suite de l'analyse sera faite par FUNMAP [3.4.3.9.]

. "β" n'est introduit que pour les opérandes qui sont des noms de clause. Deux cas sont possibles :

- il s'agit d'une équation en "γ"
  appel de TSTGLB [3.4.3.8.]. Seul le deuxième opérande peut être un nom de clause (le premier étant un nombre de niveau)
- appel de la routine TSTLOC [3.4.3.7.]. Les deux opérandes peuvent être un nom de clause.

3.4.3.5. COPY

Routine standard qui recopie d'un vecteur dans un autre, une zone de longueur déterminée.

3.4.3.6. CLCALL (clause call)

La structure de définition des clauses a été représentée au moyen d'une arborescence binaire [3.4.1.3.]. Un opérande, défini dans une clause "C" de niveau "n" sera un nom de clause s'il est un sommet de la branche :

- dont la racine est le sommet adjacent à C (1)
- de niveau n+1 (2) [fig. 13.]

La fonction CLCALL teste si un opérande de l'équation est un nom de clause. Elle prendra donc pour valeur :

- i si l'opérande est le sommet "i"
- o si l'opérande n'appartient pas à cette branche.

La numérotation des sommets de l'arborescence est obtenue en utilisant l'algorithme de traversée suivant :

- visiter la racine
- traverser le sous arbre de droite
- traverser le sous arbre de gauche

Cette façon de traverser l'arborescence correspond à la numérotation des sommets suivant leur ordre d'apparition dans le programme. Le premier sommet (MAIN) porte le numéro "1).

La fonction CLCALL a deux arguments :

- nom d'opérande : nom dont il faut tester l'appartenance à la branche déterminée par (1) et (2)
- niveau : niveau où la variable est définie.

ex : soit l'arborescence binaire

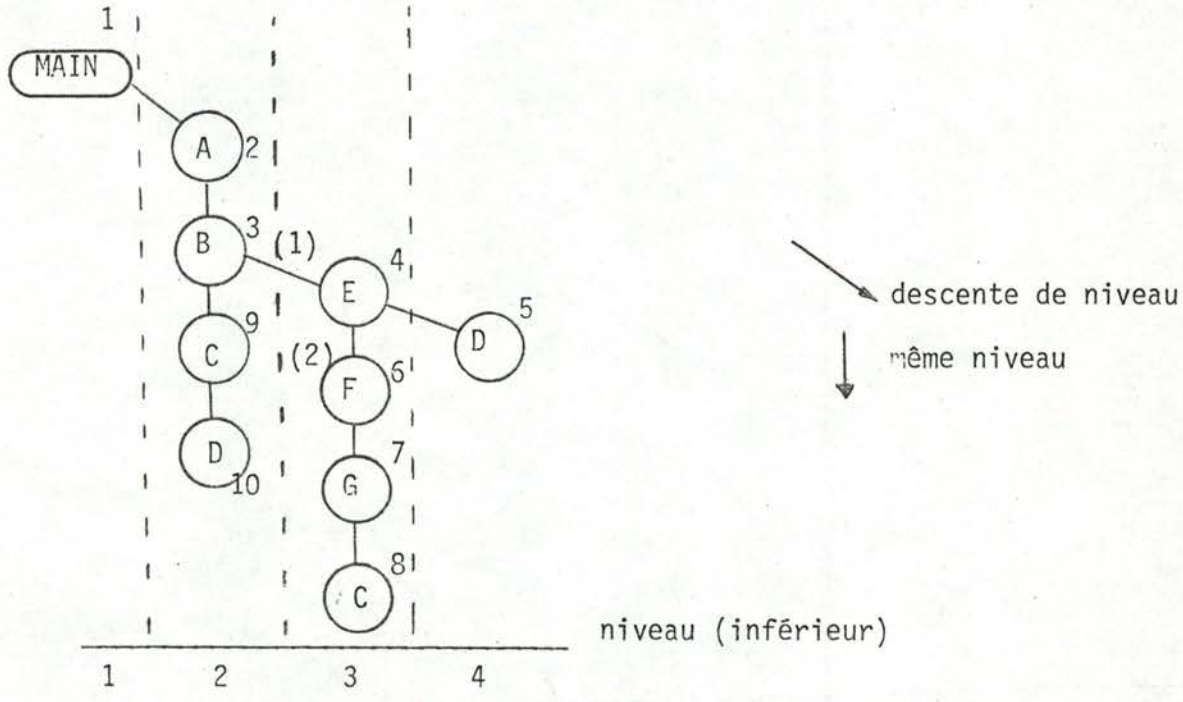


Fig. 13 : squelette d'un programme Lucid

. Analyse de la clause "A" (niveau 2)

CLCALL (D, 2) = 0 car aucun sommet de niveau inférieur accessible pour A (c'est-à-dire branche vide)

. Analyse de la clause "B" (niveau 2)

CLCALL (D, 2) = o car D $\notin$ branche obtenue par (1) et (2)

CLCALL (F, 2) = 6

3.4.3.7. TSTLOC (test if local clause)

soit"n" le niveau de la clause en cours d'analyse.

Appel de la fonction CLCALL pour chacun des opérandes de l'équation. L'opérande "β" sera introduit [3.4.3.11] pour les opérandes tel que CLCALL (operande, n)>0 c'est-à-dire pour les opérandes qui sont des noms de clause.

3.4.3.8. TSTGLB (test if global clause)

même fonction que la routine précédente mais pour des équations en "γ". "β" sera donc introduit [3.4.3.12] si CLCALL (opérande, n)>0

où . opérande = second opérande de l'équation

. n = niveau actuel - nombre de niveau à remonter (premier opérande de "γ")

= niveau de la clause où a été définie l'opérande.

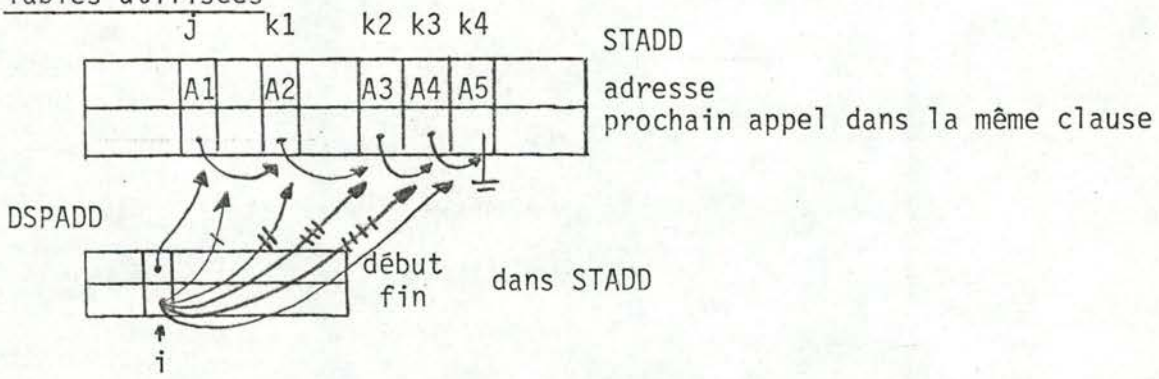
3.4.3.9. FUNMAP (function - mapping)

Nous sommes dans le cas où "β" remplace le séparateur introduit par l'analyseur lexicographique, c'est-à-dire dans le cas des appels de clauses à arguments Lorsque le nom de la clause appelée suit l'opérateur "β", nous donnons une nouvelle représentation du nom de la clause. (Ce ne sera vrai que pour les appels de clauses locales à la clause analysée). Dans le cas contraire, le nom de la clause sera rebaptisé lors de son occurrence (dans une équation du type "γ" ou "cons

3.4.3.10. UPDCALL (update call address)

La routine est appelée lors de chaque appel de clause; elle mémorise l'adresse de l'appel (numéro de ligne)

Tables utilisées



La zone contenant les adresses des différents appels possibles n'est pas nécessairement continue : ceci est dû à la structure d'imbrication.

Dans l'exemple, pour la clause de nom "i", la chaîne des adresses d'appels commence en "j" et se termine en "k4"

3.4.3.11. LOCCL (local clauses)

La routine introduit l'opérateur "β", pour des appels de clauses locales. Le nombre d'appel à l'intérieur de la clause est incrémenté de 1
Un avertissement sera émis pour le passage de "function" ou de "mapping" en tant qu'argument. Ceci complique en effet le contrôle de concordance entre le nombre de paramètres formels et actuels. Le contrôle, dans ce cas particulier sera laissé aux soins du programmeur !
"β" ne sera pas introduit lorsque le nom d'une clause se trouve dans une équation en "cons" car dans ce cas, il ne s'agit pas d'un appel de clause mais d'une façon de faire passer le nom d'une clause en tant que paramètre actuel.

Remarques :

- les clauses utilisées en tant qu'argument ne gênent en rien l'interpréteur. (d'où avertissement et non erreur)
- par mesure de précaution, lors de l'évaluation d'un argument numéro i, l'interpréteur testera si "i" est bien un numéro de paramètre possible : le paramètre correspondant doit figurer dans une équation en "cons".

ex : $X = \beta f^{\star} T_1$

T_1 = cons A nill

$Y = \alpha_1$ OK

$X = \alpha_2 \rightarrow$ ERR (Le paramètre devrait se trouver dans l'équation d'adresse nil).

3.4.3.12. GLBCL (global clauses)

soit $n^{\star}$ le niveau de la clause analysée

EQUATN [3.4.3.4.] a détecté une équation du type : $X = \gamma n$ Cadix (1)

si a) CLCALL (Cadix, $n^{\star}$-n)>0

et

b) Cadix n'est pas le nom d'une clause de type "fun-map"

Alors GLBCL - introduit "β"

- demande à mémoriser l'adresse de l'appel de Cadix (= numéro de ligne de l'équation (2))

(1) $\rightsquigarrow$ $\begin{cases} X = \gamma\ n\ T \\ T = \beta\ \text{Cadix nil} \quad (2) \end{cases}$

Si "x" et "n" sont tel que . CLCALL (x, n) > 0

. x est le nom d'une clause "fun" ou d'une clause "map"

Alors, l'équation en "γ" ne sert qu'à transmettre le nom de la clause. Ceci provient de l'éclatement d'une équation d'appel à une clause globale en deux

équations : . équation en "β"
. équation en "γ"

Un problème subsiste pour les clauses globales à certains types de clauses.
Considérons l'exemple :

```
X = ...
fun f(A) usg X
    out = X                          P
end
cmp C usg X, f
    E = ...
    RES = X + f(E)        (3)
end
```

Dans notre cas, f est une fonction constante. L'équation (3) peut donc s'écrire

RES = X + X (4)

mais nous ne pouvons écrire RES = 2 * X

En effet, soit l'historique de X comme étant N^+, c'est-à-dire X = 0, 1, 2, ...
Dans ce cas, (3) vaut (si on fait référence à, l'instant t=1, à C) :

RES = X + f(E) = 1
↓ ↓
0 1

Nous avons dû employer latest pour X, variable globale de C mais non pour X, variable globale de f. (car variable globale d'une fonction)

Nous avons donc dans une même équation [(4)] deux occurrences d'une même variable "X" mais représentant chacune des choses différentes.
L'écriture de l'équation (4) va donc à l'encontre de l'esprit Lucid.

Définition :
Une clause est dite de type T si :
. elle est de type "fun" ou "pro"
et
. elle est utilisée en tant que clause globale dans une clause "cmp" ou "map"

La difficulté soulevée plus haut, nous amène à rejeter l'emploi de clause de type T.
En fait, la restriction est trop forte : le problème ne se pose que lorsque les clauses de type "T" ont des variables globales communes avec ces clauses "cmp-map".
Malgré cette rectification, la restriction reste-t-elle légale ? Car en agissant de la sorte, nous avons peut être restreint les possibilités du langage.
Il faudrait pouvoir démontrer que dans tous les cas il est possible d'écrire un programme, soulevant ce problème, d'une autre façon et n'achoppant pas contre cette restriction.

Dans notre cas, le programme P peut s'écrire :

```
X = ...
cmp C
   RES = latest X + X
end
```

$\Longrightarrow$

```
X = ...
C = X + next X
```

Malheureusement, nous ne pouvons dire à ce jour, que ce sera toujours possible. La restriction reste, mais n'est pas légalisée !

"Those that fly may fight again,
which he can ne'er do that's slain."

"Qui fuit pourra combattre encore,
Et ne le pourra qui est mort."
(Samuel Butler)

3.5. Nouvelle "représentation" des variables (ou "le rebaptiseur")

- L'analyseur lexicographique s'est chargé de représenter les variables par un nombre $\geqslant 63$. Ceci était nécessaire car une équation étant de taille variable, (elle peut contenir plusieurs opérateurs, plusieurs opérandes) il fallait pouvoir différencier un opérateur d'un opérande. Ce n'est plus le cas pour les équations Lucodes : celles-ci ont un format constant.

  Var [=] opérateur opérande1 opérande2 (1)

  Le signe d'égalité n'est pas repris dans la représentation interne.

Il est donc possible, de par la position, de dire s'il s'agit d'une variable ou d'un opérande. Il n'est donc plus nécessaire de distinguer leur représentation.

- Par le principe d'évaluation énoncé en [1], l'interpréteur en lisant l'équation (1) va devoir - évaluer opérande 1
  - évaluer opérande 2 (s'il existe)
  - faire agir l'opérateur sur la valeur des deux opérandes

Le résultat obtenu sera la valeur que prendra "Var".
L'interpréteur doit donc être capable d'accéder aux équations définissant les deux opérandes.
Nous allons nous arranger pour que la définition d'une variable d'une clause soit rangée à l'adresse "a" de la zone mémoire allouée à cette dernière, où "a" est le nombre représentant la variable.
Etant donné la structure d'imbrication, un même nombre peut représenter des variables de signification différente . Le "rangement" des équations se fera donc clause par clause.

Adresse d'une variable

- Le Lucode final sera produit dans des vecteurs : un pour les opérateurs, un pour les premiers opérandes et un pour les seconds. L'adresse d'une variable sera l'indice tel que les éléments correspondants dans les différents vecteurs reconstituent l'équation définissant cette variable.

  Note : la notion d'indice sera confondue dans ce qui suit avec "numéro de ligne".

- Ayant gardé la structure d'imbrication, nous parlerons d'adresse de variable en tant "qu'adresse relative" : définie par rapport au début de la clause.

De la façon dont les variables ont été identifées, il est possible qu'une variable soit représentée par un grand nombre, soit "n", alors que la clause ne contient qu'un petit nombre d'équations. Ceci implique qu'il faudra, pour cette clause, une zone ayant au moins n lignes pour pouvoir ranger l'équation définissant la vairable citée plus haut. Le taux d'occupation de la zone mémoire alloué sera donc relativement faible. Pour optimiser ce taux d'occupation au maximum, il suffit de rebaptiser les variables suivant leur ordre d'apparition à l'intérieur de la clause qui les contient.

Les équations seront ensuite réordonnées : une équation sera "rangée" à la ligne l où l est le nombre représentant la variable de partie gauche.
Il faut également rebaptiser les variables de telle sorte que les appels de clause se regroupent [3.4.3.] . Ces équations seront rebaptisées en premier lieu mais il faut cependant être prudent.
Considérons à cet effet le programme Lucode (première forme) suivant :

numéro de ligne		
1	X [=] + Y Z	P
2	Y [=] β F_1^* A	
3	A [=] cons S nil	
4	S [=] ...	
5	Z [=] β F_2^* B	
6	B [=] cons T nil	
7	T [=]	
8	out [=]	

Rappelons que les adresses des appels ont été mémorisées dans une table lors de l'analyse syntaxique. Ceci permet de rebaptiser les équations (2) et (5) en premier lieu.

(2)⇝ V # 2 [=] β $F_1^\star$ V # 3

(5)⇝ V # 4 [=] β $F_2^\star$ V # 5

où . V#i = variable numéro "i"

. V#1 = "output" par convention

Les appels de clauses ne seront par regroupés : la définition de "V#3" viendra en effet s'intercaler entre les deux appels de clauses après avoir rangé les équations correspondantes suivant le critère d'ordre énoncé plus haut.

Il faudra rebaptiser les appels de clause suivant l'algorithme :

1) rebaptiser la partie gauche de tous les appels à l'intérieur d'une clause

(2)⇝ V # 2 [=] β $F_1^\star$ A (2 bis)

(3)⇝ V # 3 [=] β $F_2^\star$ A (3 bis)

2) rebaptiser la partie droite de ces équations

(2 bis)⇝ V #2 [=] β $F_1^\star$ V # 4

(3 bis)⇝ V #3 [=] β $F_2^\star$ V # 5

Les appels de clauses rebaptisées, nous pouvons rebaptiser le reste du programme puis ordonner les équations suivant le critère cité plus haut. Ceci produit le programme Lucode :

n° de ligne						Ancien n° de ligne
1	V	# 1	[=]	.....		(8)
2	V	# 2	[=] β	$F_1^\star$	V # 4	(2)
3	V	# 3	[=] β	$F_2^\star$	V # 5	(5)
4	V	# 4	[=] cons	V # 7	nil	(3)
5	V	# 5	[=] cons	V # 8	nil	(6)
6	V	# 6	[=] +	V#2	V#3	(1)
7	V	# 7	[=]	.....		(6)
8	V	# 8	[=]	.....		(7)

Notons que la partie gauche des équations devient redondante avec le n° de ligne. Elle ne sera donc plus reprise dans la représentation du Lucode final. Le résultat de cette dernière phase du transformateur est un programme désarticulé : le programme devient un puzzle dont chaque élément est la définition d'une clause. Il est possible partant d'une clause quelconque de reconstituer tout le programme en se servant des équations en "γ" et "β".

Nous allons exposer la dernière étape du transformateur en deux étapes :

- nouvelle identification
- nouvel ordre des équations

Note : . Le corps d'une clause est délimité par une en-tête et une fin de clause. Elle peut contenir d'autres définitions de clauses et donc d'autres corps de clauses.

. La définition d'une clause est l'ensemble des équations définissant les variables locales (non les clauses locales)

3.5.1. Nouvelle identification

Cette phase a pour fonction de :

- donner un nouveau nom aux variables
- mémoriser les limites de la définition d'une clause.(Cette dernière ne constitue pas toujours une zone continue en mémoire.)

3.5.1.1. RENDRI (rename driver)

Aiguillage selon le type de phrase

3.5.1.2. RENCAL (rename clause calls)

Rebaptise les appels de clauses suivant l'algorithme déjà cité

3.5.1.3. RENVR (rename variable)

Rebaptise une variable

3.5.1.4. RENHD (rename clause head)

- Rebaptise les variables globales propres à la nouvelle clause
- Rebaptise la variable output de la clause.
  Cette variable (RES-OUT suivant le type de clause) doit rester la variable n° 1 pour chaque clause.
- demande à rebaptiser tous les appels à l'intérieur de la clause
- supprime l'en-tête de la clause
  Rappelons que l'en-tête n'est plus porteuse que d'une seule information : elle délimite la définition d'une clause; la notion de variable globale et de paramètre formel ayant été traduite au moyen d'équations en "γ" et "α"

3.5.1.5. ENDCL (end of clause)

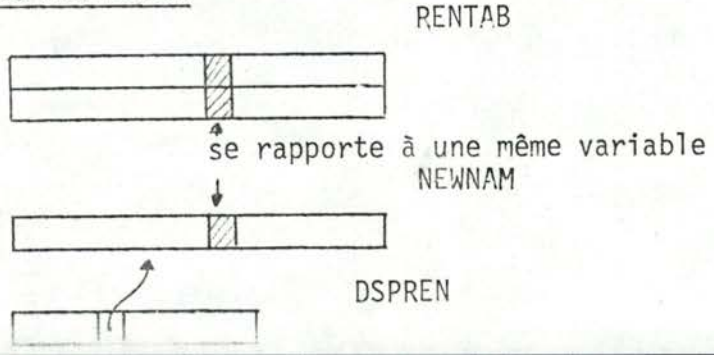
- supprime la fin de clause (celle-ci ne servait que comme délimiteur)
- détecte les variables définies plus d'une fois à l'intérieur d'une même clause (erreur)

3.5.1.6. ADELE (add element)

Donne l'ordre d'apparition de la variable à l'intérieur d'une clause. Pour ce faire, elle mémorise dans une table les variables déjà rencontrées.

Chaque clause aura ainsi sa table d'identification.

Tables utilisées



. DSPREN : donne le début de la table d'identification propre à une clause

. NEWNAM : contient les différentes tables d'identification suivant un système de stack.
Une table d'identification contient les variables déjà apparues dans la clause.

. RENTAB (1,) → 0 variable non définie (erreur)
→ 1 variable définie une et une seule fois
>1 variable définie plus d'une fois (erreur)

. RENTAB (2,) → 0 variable non utilisée
→ 1 variable utilisée au moins une fois.

3.5.1.7. ADNAM (add name)

Routine appelée lors d'une nouvelle définition de clause.

Elle introduit une nouvelle table d'identification et l'initialise avec la variable output de la clause (res-out suivant le type de clause)

3.5.1.8. ADGLB (add globals)

La routine rebaptise les variables globales propres à une clause. Seules les variables globales pour un niveau le seront (les autres l'ayant été précédemment) ADGLB est appelée lors de la définition d'une clause.
Il suffit donc, avant d'introduire une nouvelle table d'identification, de rajouter, s'il y a lieu, ces variables dans la table d'identification de la clause précédente (= la table précédente car système de "stack")

3.5.1.9. PHBGOB photo beginning of block

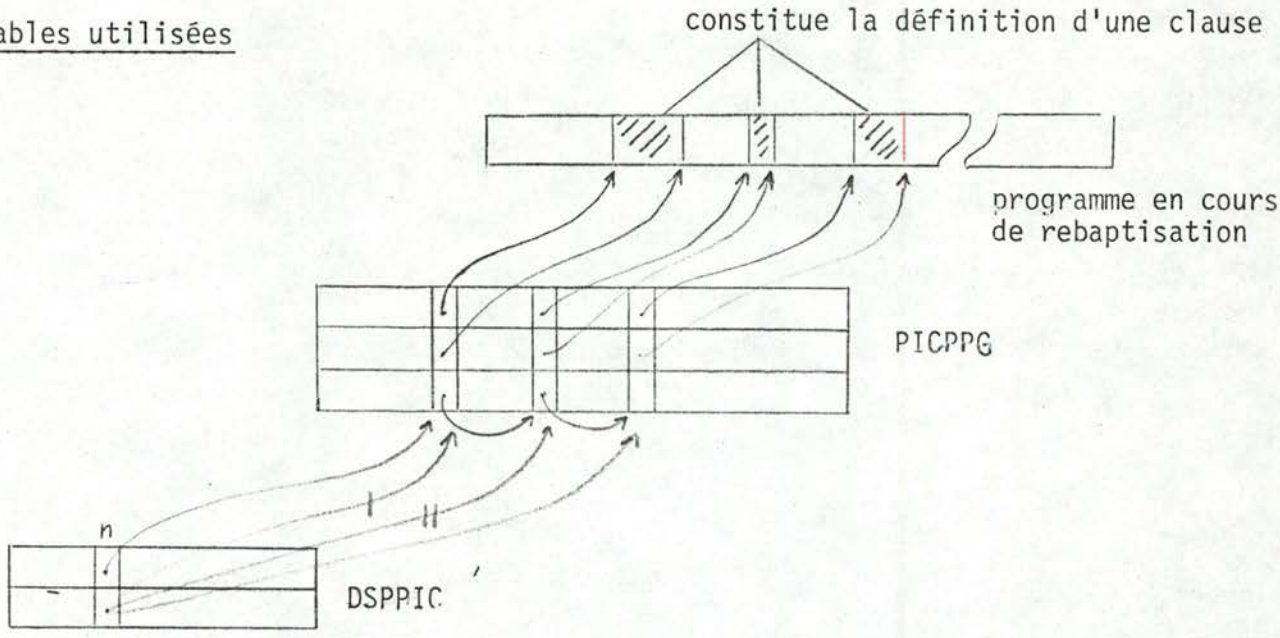
3.5.1.10. PHEDB photo end of black

"Die Daguerreotypie ist ein Zeugnis gegen die irrige Ansicht dass die Kunst eine Nachahmung der Natur sei.
La photographie démontre qu'il est faux de croire que l'art est une imitation de la nature."

(Heine)

Ces deux routines mémorisent les différentes limites de la définition d'une clause.

Tables utilisées



. DSPPIC(1, n) : donne le début de "l'image" de la clause "n" dans PICPRG
. DSPPIC(2, n) : en donne la fin courante
. PICPRG(1, j) : fournit le début d'une "partie" de clause
. PICPRG(2, j) : fournit la fin de ce morceau
. PICPRG(3, j) : permet de trouver la suite de "l'image" dans PICPRG

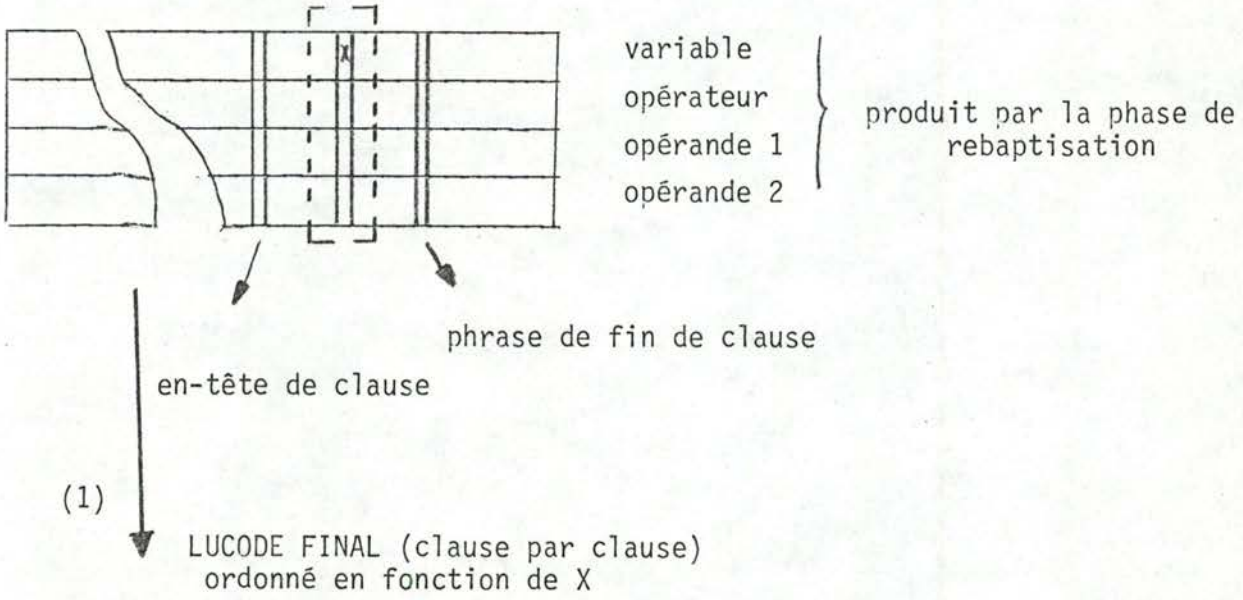
PICPRG : élément chaîneur

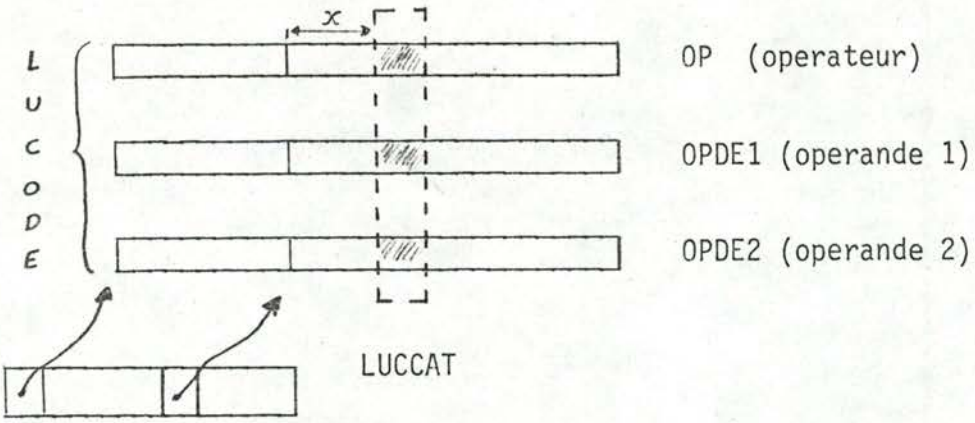
3.5.2. Nouvel ordre des équations

3.5.2.1. TRSDRI (transformer driver)

Il s'agit du "programme principal" : il déclenche tout le processus de transformation.

Il ordonne les équations clause par clause en suivant le procédé déjà énoncé, mais que nous rappelons : "l'équation définissant la variable "j" sera rangée à la ligne "j" de la zone correspondant à cette clause".





LUCCAT : Ce catalogue permet, pour un nom de clause donné, de retrouver l'équation définissant sa variable output.

L'ordre dans lequel nous avons rangé les clauses (1) importe peu. Nous avons gardé celui de l'algorithme de traversée de l'arborescence squelette. [3.4.3.6.]

3.5.2.2. LIMIT

fournit à TRSDRI les différentes limites de la définition d'une clause. Celles-ci sont connues au moyen de tables [3.5.1.10.]

3.5.3.2. CONCOR : vérifie le nombre de paramètres

Pourquoi avoir retardé cette vérification jusqu'ici ?
L'ordre des équations ou des définitions de clauses importe peu au sein du corps d'une clause (nous considérons un programme Basic Lucid comme étant une clause).

Lors de la lecture d'un programme Lucid, la définition d'une clause n'est donc pas nécessairement connue lors de son appel. La vérification ne peut donc se faire qu'après la lecture du programme tout entier. Ce fut le cas lors de l'introduction de l'opérateur "β", mais la phase précédente avait fait éclater une équation en plusieurs autres si bien que cette fois, le nom de la clause appelée n'était plus nécessairement directement connu dans l'équation de l'appel. L'accès à l'équation contenant ce nom est facile à ce stade car la représentation d'un nom de variable et l'adresse de l'équation la définissant ne font qu'un.

La phase de transofrmation terminée, TRSDRI construit l'interface nécessaire pour pouvoir passer à l'interprétation.
Il sauve dans un fichier temporaire :

- les trois vecteurs Lucode : OP, OPDE1, OPDE2
- le catalogue Lucode (LUCCAT)
- le catalogue des clauses (CLCAT)

Pour rappel, CLCAT contient pour une clause donnée :

- son type
- le nombre d'appels qu'elle contient
- l'adresse du premier appel

3.6. Conclusion

La phase de transformation a consisté à :

- introduire les opérateurs α, β, γ
- décomposer une équation à plusieurs opérateurs en plusieurs équations à un opérateur
- rebaptiser les variables de telle sorte que le nombre les représentant soit aussi l'adresse de l'équation les définissant.

Les différentes entités sont représentées par des nombres.

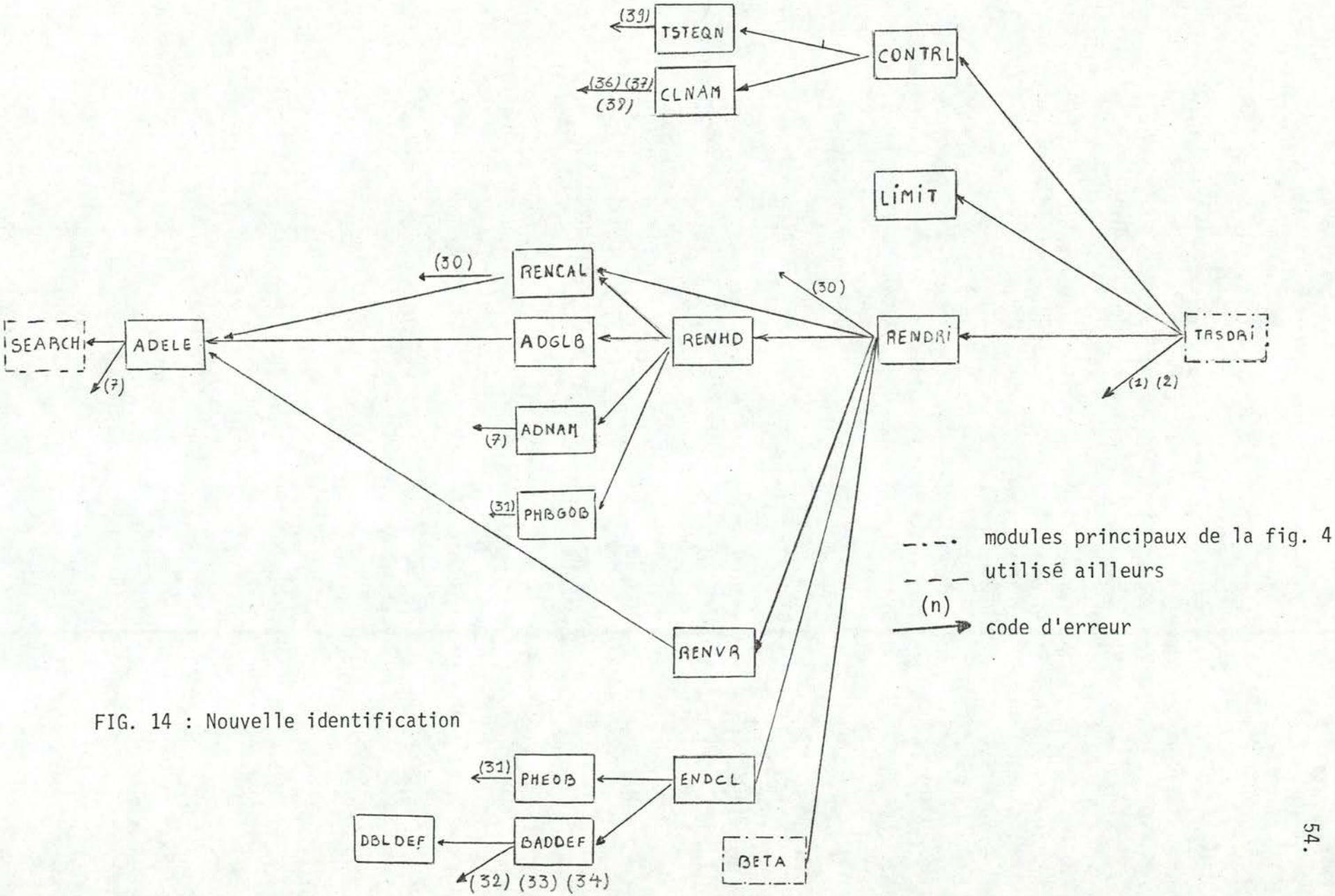


FIG. 14 : Nouvelle identification

§4 JUSTIFICATION DES OPERATEURS " α", "β", "γ"

Note : Ce paragraphe est fortement inspiré des recherches de Mr Wadge.

Définition :

Environnement : fonction ρ qui associe des valeurs aux variables

ρ : var $\rightsquigarrow$ domaine de valeurs

Nous disons que :

produce P using G_0 , G_1 , G_2 , ...
Δ (ensemble d'équations)
out
end

est vrai pour un environnement ρ ssi

$\exists$ un environnement ρ' tel que

- $\rho(P) = \rho'(out)$
- $\rho(G_i) = \rho'(G_i)$ $\forall i$
- $\rho' \models E$ $\forall E \in \Delta$

Soit E de la forme "V = exp"

$\rho' \models E \iff \rho'(V) = \rho'(exp)$

La notion d'environnement pour une expression est parfaitement définie :

$\rho'(A + B) = \rho'(A) \oplus \rho'(B)$

Au cours des différentes transformations, nous avons introduit l'opérateur "γ". Il y a donc dans des équations de la forme : "variable = γ n G_i" (1)

La notion d'environnement a été introduite dans le formalisme Lucid et peut donc être appliquée à toute expression Lucid. Le Lucode a cependant introduit de nouvelles expressions : en α, β, γ

Dans le cas qui nous préoccupe ici, nous allons étendre la notion d'environnement aux expressions en "γ".

1) Structure d'imbrication d'un niveau seulement (n=1)

$\rho \models$ produce P using G_0 , G_1 , G_2
Δ
end

ssi $\exists$ ρ' tel que 1) $\rho(P) = \rho'(out)$
$\rho(G_i) = \rho'(G_i)$
$\rho\rho' \models E$ $\forall E \in \Delta$

où $\rho\rho'$ est défini de la façon suivante :

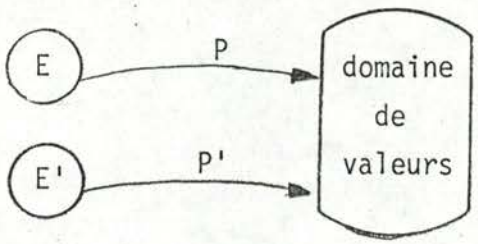
- pour une expression Lucode en γ : $\gamma_1 X$

$\rho\rho'(\gamma_1 X) = \rho(X)$

- pour une expression Lucid : A + B

$$\rho\rho'(A + B) = \rho'(A + B) = \rho'(A) \oplus \rho'(B)$$

Que représente $\rho\rho'$?



E = ensemble des variables locales d'une clause

E' = ensemble des variables locales d'une clause imbriquée à la première

$\rho\rho'$ = fonction définie sur E U E'

2) Généralisation

soit $\rho = \rho_0 \rho_1 \ldots \rho m$

$\rho \models$ [produce P using G_0 , G_1 , G_2
Δ
end]

ssi ∃ un environnement σ tel que :

1) $\sigma(G_j) = \rho(G_j)$

2) $\sigma(out) = \rho(P)$

3) $\rho_0 \rho_1 \ldots \rho_m \sigma \models E \qquad \forall E \in \Delta$

Ceci nous permet d'affirmer du point de vue formel que :

produce P using G_0 , G_1 , G_2 ...
Δ
end

est équivalent à : pro P
Δ'
end

où Δ' est obtenue en remplaçant toute occurrence d'un G_j par $\gamma n G_j$

Une théorie analogue a été développée par Mr Wadge pour les opérateurs "α" et "β"

Conclusion

La phase de transformation a produit un programme Lucode équivalent, du point de vue formel, au programme Lucid. Ils auront ainsi les mêmes valeurs à l'output.

§5. L'INTERPRETEUR

Rappelons qu'un interpréteur existait déjà pour le Basic Lucode. Nous avons :

- étendu cet interpréteur aux clauses
- amélioré ses performances.

5.1. Interpréteur du Basic Lucode

Son mode de fonctionnement est décrit dans [1]. Rappelons-le brièvement. Le mécanisme consiste à demander la valeur d'une variable à un instant t (t n'est pas à prendre comme le temps horloge). Le processus est déclenché en demandant la valeur, à un instant t, de la variable output du programme. Cette demande générera automatiquement d'autres demandes.
Il s'agit d'une succession de "call by need"[§2]
L'interpréteur est composé d'une routine chargée d'évaluer les opérandes d'une équation : l'évaluateur "EVAL" [fig. 15.]. Ce dernier a été écrit en reprenant les sémantiques des différents opérateurs pouvant être utilisés en Basic Lucid.
A chaque code opératoire correspond un traitement particulier.
Ainsi lors de l'évaluation de "V(t)" définit comme étant :

"V = A + B", EVAL - évalue A(t)
- évalue B(t)
- additionne les valeurs obtenues. Le résultat est la valeur de V(t)

Note :

X(t) = valeur de X à l'instant t

Rappelons que dans la configuration interne du Lucode, une variable est représentée par un nombre qui est le numéro de ligne où la variable a été définie. L'évaluation d'une variable consistera à interpréter son équation de définition.
L'évaluation se fait donc de manière récursive.

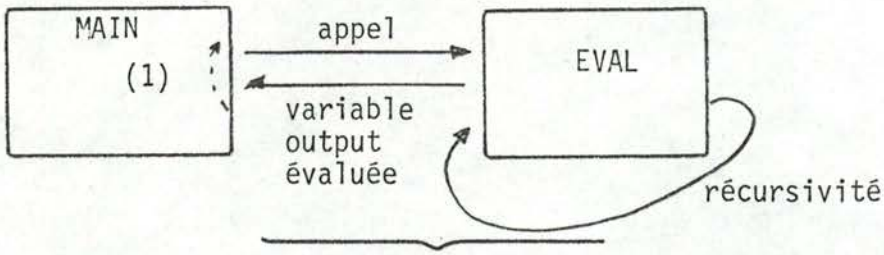


Fig. 15 : interpréteur Lucode

(1) Un programme Lucid est à considérer comme un processus itératif : il ne fournit pas une valeur mais un historique de valeurs pour sa variable output; soit output au temps t_0 , t_1 , t_2 ... Pour activer l'évaluateur, il suffit de demander la valeur de output au temps t_0 . Une fois la demande satisfaite, le processus sera répété pour t_1 , t_2 ....

Afin d'éviter que le processus ne se déroule durant un temps infini, nous nous somme limités à un historique de 10 valeurs pour la variable output du programme.

5.2. Interpréteur étendu

Les seuls nouveaux opérateurs introduit en Lucode étendu sont les opérateurs :

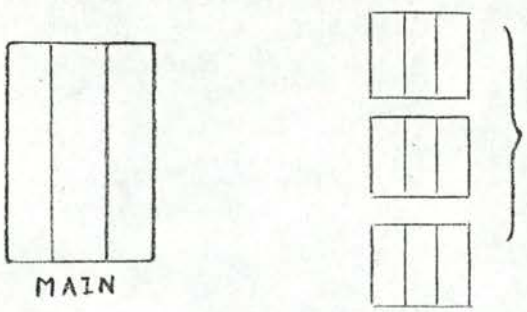
- ALPHA
- BETA
- GAMMA
- LAMBDA

Il suffit donc d'ajouter pour chacun de ces opérateurs, le traitement correspondant à faire.

5.2.1. Le graphe traceur

"Ainsi chaque bon arbre apporte-t-il de bons fruits."
(Saint Mathieu)

Le transformateur a produit un programme Lucode sous la forme :

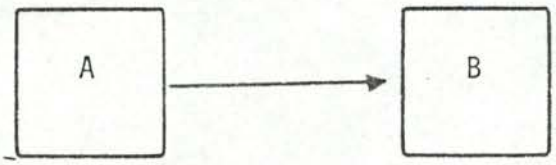


L'évaluation de l'output du programme : - active le Main
- va générer plus tard des appels aux différentes clauses.

Un appel de clause revient à "ajouter" celle-ci à l'ensemble des clauses déjà actives, d'où l'idée, primitive, de recopier le corps de la clause lors de son appel. Cette idée fut quelque peu développée avant l'introduction des opérateurs

"α" et "γ". L'interpréteur se construisait un"programme linéaire" en rebaptisant les variables de la clause appelée lors de la recopie. Système assez lourd donc mais qui nous ramenait, après la recopie, à l'interprétation d'un programme en Basic Lucode.

Nous allons plutôt utiliser des pointeurs vers le texte du programme Lucode afin de pouvoir utiliser toujours le même code, quelle que soit le nombre d'appel.
Soient deux clauses "A" et "B".
Que faudrait-il faire lorsque "A" appelle "B" ?



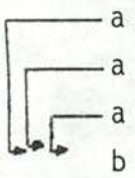
Il faut établir un lien entre "A" et "B"
Une façon élégante de gérer les appels de clauses semblerait être l'utilisation d'un stack contenant les adresses d'appels. Un sommet représenterait donc un appel. La clause en cours d'interprétation aurait son adresse d'appel au sommet du stack. L'évaluation d'un paramètre actuel consiste à revenir à la clause précédente. Ceci est possible grâce à l'adresse au sommet. L'information n'est cependant pas suffisante. L'adresse d'appel permet de savoir où débute la chaîne des paramètres actuels (deuxième opérande d'une équation en "β".) Il faut encore trouver la ligne où ce paramètre a été défini : ce numéro de ligne est disponible dans la chaîne mais, n'est défini que par rapport au début de la clause. Il faut donc savoir où débute la clause.
Cette information est disponible dans le catalogue LUCCAT [3.5.2.1.], moyennant la connaissance du nom de la clause.
Un sommet du stack devra donc au moins contenir :

- l'adresse de l'appel
- le nom de la clause appelée.

Que se passe-t-il lors de l'évaluation d'une variable globale ? Dans le cas d'appel non récursif, il suffit de revenir à la clause appelante pour la "globalité" d'un niveau et de répéter le processus "n" fois s'il s'agit d'une variable globale de "n" niveaux. Par contre dans le cas récursif, il faut revenir à la clause appelant cette routine récursive. Un sommet du stack devra donc contenir une troisième information :

- adresse du sommet lexicographique précédent

ex :



Un sommet du stack contient donc des pointeurs arrières dans le stack. Ces informations ne suffisent pas encore. Il se peut que l'évaluation d'un paramètre ou d'une variable globale fasse appel à une nouvelle clause. Ceci entraîne que les sommets correspondant à des appels imbriqués ne sont plus consécutifs. Ceci nous amène à introduire un nouveau pointeur :

1) du sommet correspondant au dernier appel vers celui de l'appel précédent. Ce pointeur permet, une fois l'appel satisfait de revenir à l'appel précédent. (pointeur arrière)

Une équation, étant un processus itératif (elle est vraie à tout instant), peut faire appel à une même clause mais à des temps différents. Il n'y a pas lieu pour autant d'introduire un nouveau sommet pour chacun de ces appels. Tout comme une variable a un historique de valeurs, un sommet doit pouvoir satisfaire l'appel d'une clause à des temps différents. Il faudra donc garder trace du premier appel dans le temps. Nous allons introduire un nouveau pointeur dans le sommet courant du stack :

2) un pointeur du sommet courant (sommet correspondant à l'appel de la clause en cours d'évaluation) vers le nouveau sommet introduit. (pointeur avant) Il s'agit de l'inverse de 1)

Ce phénomène peut se répéter pour plusieurs appels de clauses au sein d'une clause. Le pointeur avant 2) ne sera donc pas unique. Il faudra prévoir dans le sommet de la place suffisante pour les "pointeurs avants" de tous les appels possibles à l'intérieur d'une clause. Les sommets ne seront plus tous de même taille.

D'après les types de liens entre les éléments du stack, il n'est plus possible de parler de stack mais bien d'un arbre ou plutôt d'un graphe (car 1) et 2) forment un cycle).

Remarque :

L'idée d'un stack pouvait-elle répondre aux objectifs du Lucid étendu ?
Le sommet d'un stack monte ou descend de niveau à la suite de certains événements.
Dans notre cas ce serait : - nouveau sommet lors d'un appel
- suppression de ce sommet lorsque l'output est évalué.

Les clauses "pro-fun" sont considérées comme des processus itératifs, c'est-à-dire actifs durant un temps infini. Supprimer un sommet revient donc à supprimer les liens entre clause appelante et clause appelée; c'est-à-dire désactiver cette dernière (elle ne saura plus évaluer ses variables globales ...). L'idée du stack s'oppose à la fonction de ce type de clauses.

La suppression d'un sommet du stack est cependant légale dans le cas des clauses "cmp-map" (clauses non itératives). Elle pose cependant des problèmes d'ordre pratique. Les sommets, correspondant à des appels imbriqués, ne sont pas nécessairement consécutifs. La suppression d'un sommet peut donc se faire au milieu du stack. La récupération de la zone ainsi libérée nécessiterait une mise à jour de pointeurs. Plutôt que de supprimer, dans le graphe, le sommet correspondant à un appel de clause "cmp-map"; nous allons le garder afin de pouvoir le réutiliser lors d'un appel ultérieur.

Ce graphe permet, à tout instant, de voir toutes les "connexions" entre les clauses actives d'où le nom de graphe traceur.

Description d'un sommet du graphe

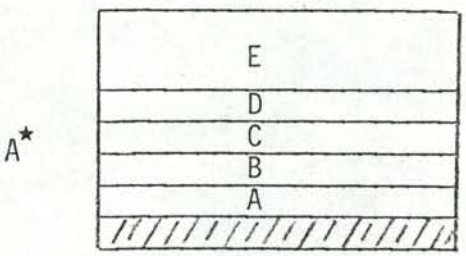


Fig. 16.

A : pointe vers le sommet précédent

B : pointe vers le sommet lexicographique précédent

C : nom de la clause appelée

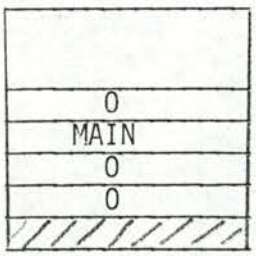
D : adresse de l'appel (= numéro de ligne par rapport au début du programme ⇒ "adresse absolue")

E : contient les pointeurs vers les sommets correspondant aux appels contenus dans cette clause.
dimension de E = nombre d'équations d'appel que la clause contient

Il existe un ordre sur E. Nous venons de voir que dans le cas de clauses itératives, le même sommet devait être utilisé pour des appels à des temps différents.

La seule façon de tester si un sommet existe déjà est de voir s'il existe déjà un pointeur d'appel vers lui. Ceci est faisable si, ce pointeur se trouve à un endroit fixe dans E. Ce sera le cas en utilisant un "numéro d'ordre d'appel" (ordre d'apparition dans le programme et non lors de l'exécution).
Rappelons que des informations ont été mémorisées à cet effet [3.4.3.]
Soient . E^* le début de E
. n le numéro d'appel
L'adresse du nouveau sommet sera mémorisé en E^*+n

Initialisation du graphe



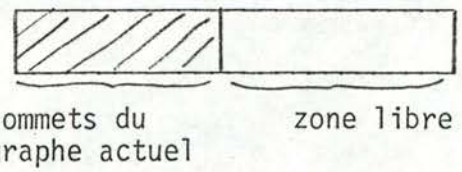
pas d'appel du MAIN

aucun pointeur arrière

MAIN = la "clause" de plus haut niveau

Représentation interne du graphe

Le graphe est représenté par un vecteur.
Un sommet est ajouté au début de la zone libre.



"Adresse d'une variable"

L'adresse d'une variable est l'adresse (numéro de ligne) où elle a été définie Ce numéro de ligne est défini par rapport au début de la clause (numéro de ligne relatif). Le numéro de ligne absolu sera :

ce numéro de ligne + LUCCAT (nom) où nom = partie C du sommet courant.

Note : LUCCAT (nom) sera appelé la base de la clause "nom".

5.2.2. Signification opératoire de "α", "β", "γ", "λ"

1. L'opérateur"α"

Rappel

format d'une équation en "α"

[V =] α n n = numéro du paramètre formel

Le deuxième opérande de l'équation, dont l'adresse absolue est la partie D du sommet courant, nous donne l'adresse relative de la chaîne des paramètres actuels. L'adresse absolue sera connue si la base de la clause appelante est connue.

Le rôle de "α" est de revenir au sommet précédent et donc à la clause appelante

La base sera obtenue par LUCCAT(NOM) où NOM = partie C du sommet obtenu.

Connaissant l'adresse absolue, il est possible de prélever l'adresse du $n^{ème}$ paramètre (première composante du $n^{ème}$ élément de la chaîne). Ayant la définition de ce paramètre, il est donc possible de l'évaluer.

2. L'opérateur "γ"

Rappel

format d'une équation en "γ"

[V =] γ n X X = nom d'une variable
n = nombre de niveau à remonter

Le rôle de "γ" est de fournir le bon environnement : celui dans lequel où X a été défini. Il permet d'accéder à la clause située n niveaux plus haut.

Il suffit de suivre la chaîne, définie au moyen de la partie B des sommets, sur une longueur de n. La base de la clause contenant la définition de cette variable globale est obtenue par l'intermédiaire de la partie "C" du sommet ainsi obtenu.

3. L'opérateur "β"

Rappel

format d'une équation en "β"

[V =] β F C F = nom d'une clause ou variable temporaire qui, après évaluation, contiendra un nom de clause.
C = adresse du début de la chaîne des paramètres actuels.

"β" ajoute, s'il y a lieu, un nouveau sommet au graphe. Il active, si ce n'est déjà fait, la clause appelée.

L'interpréteur se chargera ensuite d'évaluer l'output de la clause appelée.

L'interpréteur possède-t-il toutes les informations nécessaires ?

[fig. 16.]

- sommet courant
  - . E il faut mettre en $E^{*}+n$, l'adresse du nouveau sommet.
    n = numéro d'appel = adresse de l'appel - adresse du premier appel dans cette clause + 1 [3.4.3.]
    adresse du nouveau sommet = adresse du début de la zone libre [5.2.1.]
- nouveau sommet
  - . A : adresse du sommet que nous venons de quitter
  - . B : - dans le cas de clause locale : même chose que A
    - dans le cas de clause globale
      - . suivre la chaîne lexicographique sur une longueur l (= premier opérande de l'équation en "γ")
      - . recopier l'élément B de ce sommet.

    Dans le cas particulier où le sommet obtenu est le sommet initial, l'élément B sera l'adresse du sommet initial (=1 étant donné la représentation du graphe)
    - dans le cas de nom de clause en tant que paramètre
      - . recopier l'élément B du sommet précédent
  - . C : Le nom de la clause appelée est le premier opérande de l'équation d'appel. S'il ne s'agit pas d'un nom, il faut évaluer ce premier opérande. Le nom d'une clause est une constante, donc $\geqslant 500$.

4. L'opérateur "λ"

L'opérateur "latest" définit en Basic Lucid ne peut être implémenté comme tel car il nécessite une infinité de paramètres. Il faudra simuler son mécanisme au moyen d'une pile.

Celle-ci contiendra les différents instants où le temps a été gelé. Contrairement au Basic Lucid, l'opérateur latest ne peut être utilisé par le programmeur. Il intervient d'une façon implicite au travers des clauses "cmp-map" pour les variables globales et les paramètres. Il a été introduit dans une équation par le transformateur.

Gestion de la pile (réservée à l'opérateur λ)

Lors d'un appel de clause "gelant" le temps

- l'interpréteur mémorise dans le sommet de la pile l'instant actuel et continue son évaluation avec l'instant t = o.

- lorsque la variable "result" est évaluée, il descend le sommet d'un niveau.

Pour évaluer "λ n X", il évalue "X" à l'instant t = pile(sommet-n)

5.2.3. Exemple

"In scientiis addiscendis prosunt exempla magis quam praecepta."
"Dans l'enseignement des sciences, l'exemple vaut mieux que la théorie".
(Newton)

a) Programme Lucid

```
A = 3
fun fac(n) usg fac
  out = if n not > 1 then 1 else n * fac (n-1)
end
out = fac(A)
end
```

b) Programme Lucode

out/1	=	β	fac	T2/2
T2/2	=	cons	A/3	nil
A/3	=	=	3	

MAIN

out/1	=	then	T6/6	T7/7
T2/2	=	β	T3/3	T4/4
T3/3	=	γ	1	fac
T4/4	=	cons	T10/10	nil
n/5	=	α	1	
T6/6	=	not	T8/8	
T7/7	=	else	1	T9/9
T8/8	=	>	n/5	1
T9/9	=	*	n/5	T2/2
T10/10	=	-	n/5	1

FAC

FIG. 17

LUCCAT(MAIN)=1, LUCCAT(fac)=4 (début de fac)

où [] signifie non repris dans la représentation du Lucode

T variable temporaire

V/n variable V représenté en Lucode par n (numéro de ligne (relatif) où elle est définie)

c) Evaluation de la variable "output" au temps t = o ⇒ Eval(1, 0)
out/1 | \ temps

Note : aucune équation ne modifie le temps. Nous omettrons le paramètre dans notre exemple.

La base de la clause appelée est :

"D = LUCCAT(NOM)-1" où NOM = partie C de ce sommet [Fig. 16]

EVAL(out/1 + D) D=LUCCAT(MAIN)-1 = 0

*β fac T2/2 . Appel d'une clause : un nouveau sommet est ajouté au graphe traceur

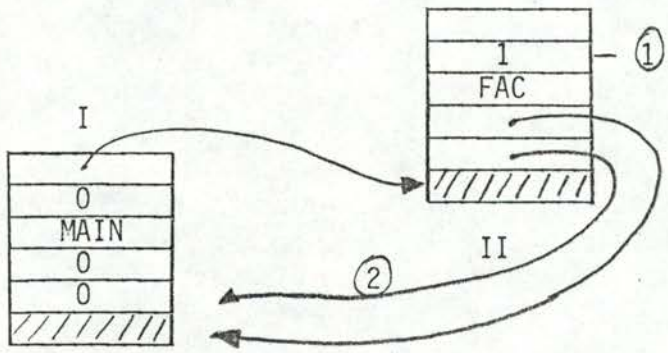


FIG. 18.

. La variable output de cette clause est évaluée. L'adresse de cette variable = LUCCAT (fac) = 4

EVAL(out/1 + D) D=LUCCAT(fac)-1 = 3
THEN T6/6 T7/7
.EVAL (T6/6+D)
NOT T8/8
EVAL (T8/8 + D)
> n/5 1
EVAL (n/5 + D)
[α 1] évaluation du paramètre actuel

. début de la chaîne des paramètres = 2ème opérande de l'équation 1 (adresse de l'appel) ①

. nous revenons au sommet précédent ② [fig. 18.]

Adresse de la chaîne = T2/2 + D = 2

où D = LUCCAT(MAIN)-1=0 [fig. 17.]

Adresse des paramètres = A/3 + D = 3

d'où EVAL (n/5 + D) ⇒ EVAL (A/3 + D)

= 3 Retour automatique grâce à la récursivité.
(Retour automatique également au sommet II)

⇒ "T8/8" = TRUE

⇒ "T6/6" = FALSE ⇒ il faut évaluer le deuxième opérande de l'équation
T7/7

. EVAL(T9/9 + D) D=LUCCAT(fac)-1 = 3

* n/5 T2/2

. EVAL(n/5 + D)

α 1

idem

= 3

. EVAL(T2/2 + D)

β T3/3 T4/4 appel d'une clause dont le nom n'est pas encore connu.

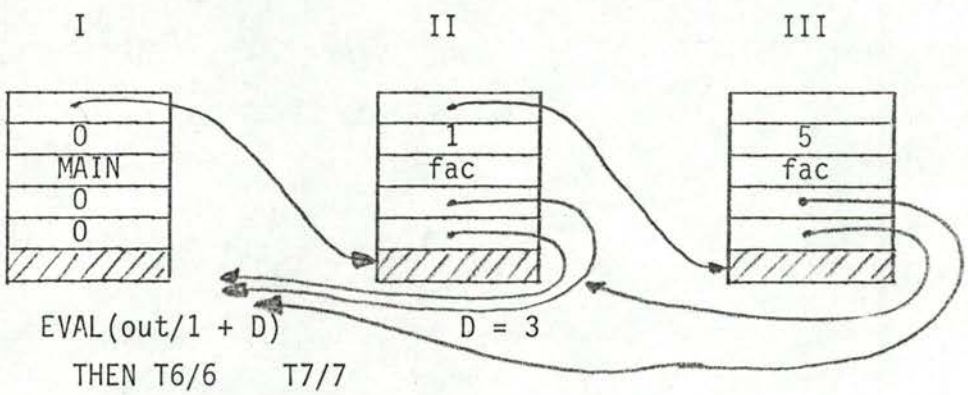
EVAL(T3/3)

γ 1 fac

L'élément B du nouveau sommet sera l'élément B du sommet obtenu en suivant la chaîne lexicographique sur une longueur de 1 (1er opérande de γ)

Nous sommes dans le cas particulier cité plus haut : sommet obtenu = sommet initial

L'élément B sera l'adresse du sommet initial.



EVAL(out/1 + D)

THEN T6/6 T7/7

EVAL(T6/6 + D)

NOT T8/8

EVAL (T8/8 + D)

\> n/5 1

EVAL(n/5 + D)

α 1

Evaluation du paramètre actuel

. Début de la chaîne des paramètres = 2ème opérande de l'équation
  β = T4/4 + D où D = LUCCAT(fac) → obtenu dans II

. Nous revenons au sommet précédent = II

EVAL(T10/10 + D)

- n/5 1

EVAL(n/5 + D)

α 1

idem

= 3

⇒ "T10/10" = 2

⇒ "T8/8" = TRUE

⇒"T6/6"=FALSE

EVAL(2ème opérande (T7/7)) = eval (T9/9 + D)

★ n/5 T2/2

EVAL(n/5 + D)

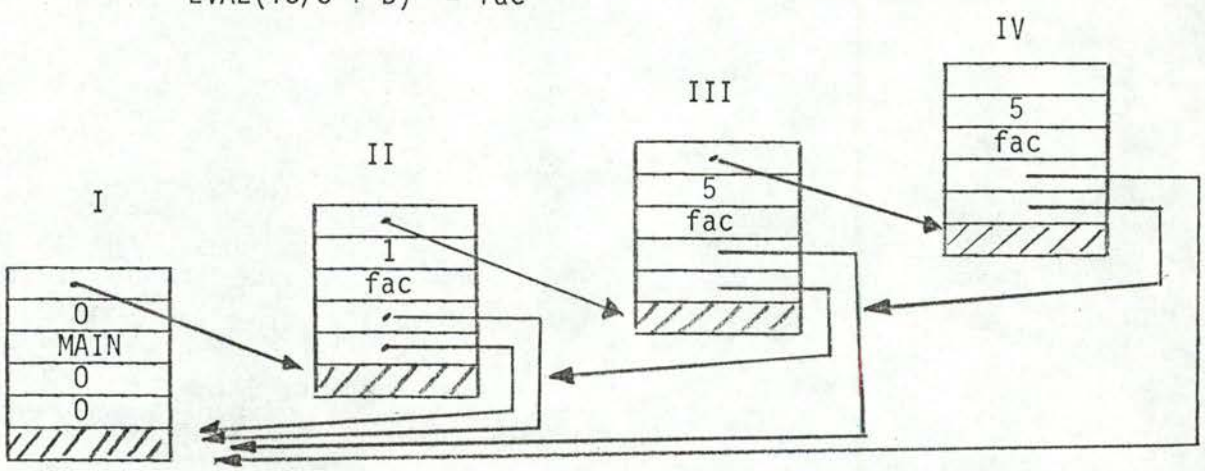
idem

= 2

EVAL(T2/2 + D)

β T3/3 T4/4 appel d'une clause dont le nom n'est pas connu

EVAL(T3/3 + D) = fac



EVAL(T6/6 + D) D=LUCCAT(fac)-1=3

NOT T8/8

EVAL(T8/8 + D)

> n/5 1

EVAL(n/5 + D)

α 1

On applique le même procédé;sommet obtenu = III

EVAL(T10/10 + D)

- n/5 1

EVAL(n/5 + D)

┌---------┐
└---------┘ idem

= 2

⇒"T10/10" = 1

⇒ "T8/8" = FALSE

⇒ "T6/6" = TRUE

⇒ "T2/2" = 1

⇒"T9/9" = 2

⇒"T2/2" = 2

⇒"T9/9" = 8

⇒"T2/2" = 6

⇒"out/1" = 6

┌---------┐ ┌---┐ idem = séquence exécutée une nouvelle fois
└---------┘ └---┘

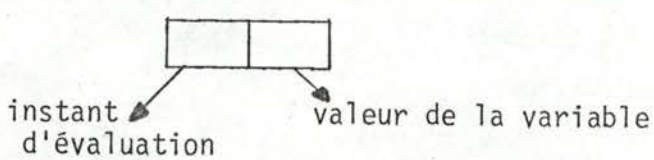
Ce système d'évaluation sera très lourd pour le calcul factoriel de grand nombre et plus généralement pour les programmes demandant plusieurs fois l'évaluation d'une même variable.
Ceci est dû à la réévaluation de variables déjà évaluées précédemment [cfr les "idem"].
Il faudrait donc pouvoir mémoriser les valeurs des variables déjà évaluées.

5.3. Interpréteur amélioré

L'interpréteur optimisé est un interpréteur étendu mais mémorisant les valeurs de variables déjà évaluées.
Une équation Lucode est telle que :

premier membre = second membre ∀t (t = temps)

L'évaluation d'une variable pour un instant t ne devrait donc se faire qu'une fois, quelle que soit le nombre de demandes. Nous allons greffer une mémoire à chaque sommet du graphe traceur. La taille de cette mémoire sera la taille (nombre d'équations) de la clause correspondant à ce sommet. A chaque variable correspondront deux éléments de la mémoire :



Lors de l'évaluation d'une variable à un instant "t*" :

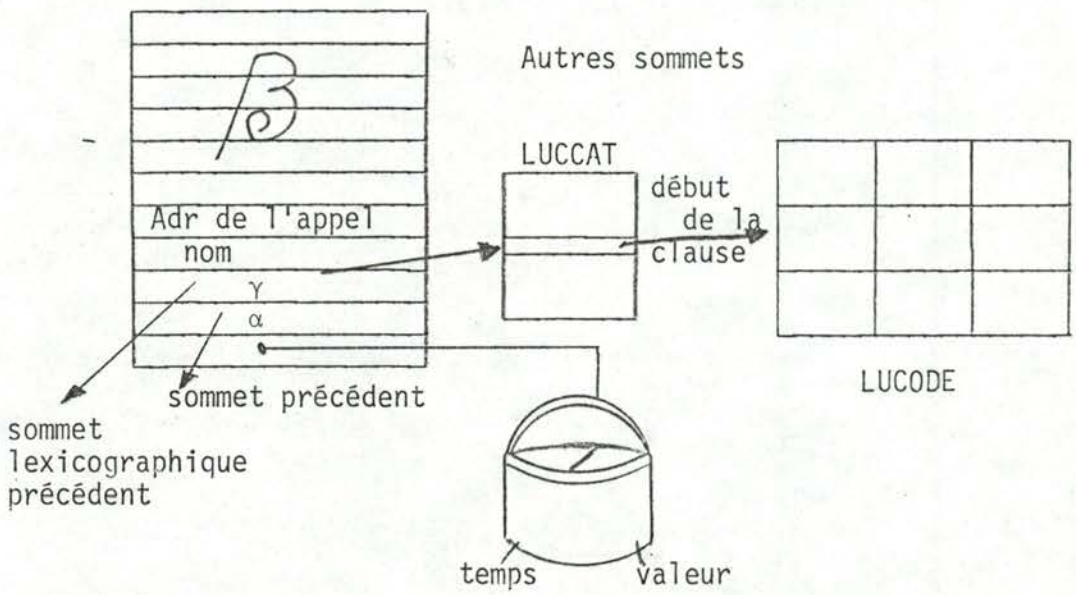
si t* est tel que la variable a déjà été évaluée pour t*, alors la valeur de celle-ci sera la valeur mémorisée.

Sinon, il faut - évaluer la variable

- mémoriser t* et la valeur de la variable

Les deux éléments mémoires correspondant à une variable ont la même adresse relative que celle-ci.

Configuration finale d'un sommet du graphe traceur



Un sommet contient : - des pointeurs dans le graphe

- des pointeurs vers le texte Lucode
- un pointeur vers une mémoire

L'optimisation temps va souvent à l'encontre d'une optimisation place mémoire, c'est pourquoi nous ne mémorisons que la dernière valeur d'une variable et non toutes celles déjà évaluées : l'évaluateur fait en effet plus souvent référence à la dernière valeur qu'à celles évaluées précédemment.

Remarque :

Une autre optimisation aurait pu être envisagée. Lors d'appels récursifs, un même sommet du graphe se duplicie (seul le pointeur vers la mémoire est modifié) autant de fois qu'il y a de niveaux de récursion. Nous pourrions ajouter un nouvel élément au sommet donnant le nombre de niveaux de récursion.

Cette méthode complique cependant considérablement l'organisation du graphe. Un sommet deviendrait en effet de taille "élastique" : dans le cas de récursion, il faudrait greffer plusieurs mémoires à un sommet.

5.3.1. Description de l'interpréteur Lucode

5.3.1.1. MAIN

. lit - le programme Lucode
- LUCCAT (catalogue Lucode)
- CLCAT (catalogue des clauses)

. initialise - le graphe
- la zone mémoire (partie temps : t = -1) [5.3.]

. active l'évaluateur en demandant : eval(1, t) t = 0, 1, ...9

variable output du programme Lucode.

5.3.1.2. ACTPAR (actual parameter)

Cherche l'adresse réelle du paramètre actuel à évaluer

5.3.1.3. CLCALL (clause call)

- Ajoute s'il y a lieu, un nouveau sommet au graphe avec sa mémoire [5.3.]
- Dans le cas clauses "cmp-map" : mémorise le temps actuel au sommet de la pile et repart avec l'instant t = 0

5.3.1.4. GLOBAL

Parcourt la chaîne lexicographique sur une longueur "L", où "L" est le premier opérande de l'équation en "γ", et dans l'adresse du sommet ainsi obtenu.

5.3.1.5. EVAL (evaluator)

EVAL(M, K) : évalue la variable, représentée par M, à l'instant "K".

Il s'agit d'une fonction récursive qui :

- évalue le(s) opérande(s) d'une équation Lucode
- applique l'opérateur sur les résultats

Avant toute évaluation, la fonction teste si la variable n'a pas déjà été évaluée

Un opérande n'est pas évalué si :

- il a déjà été évalué pour l'instant "K"
- il s'agit d'une constante.

5.4. Choix du langage d'implémentation

Nous avions trois langages à notre disposition :
- Algol
- BCPL (fort utilisé à Warwick)
- FORTRAN

BCPL a été écarté du choix pour deux raisons :

- il fallait apprendre le langage avant de faire l'implémentation
- BCPL n'existe pas ou n'est pas performant sur toutes les machines. (raison majeure)

1) Langage pour le transformateur : Le choix s'est porté sur le FORTRAN.

Vu la structure de modularité, il est souhaitable de pouvoir compiler des modules séparément. Ceci n'est pas possible sur toutes les machines utilisant l'Algol et alourdit terriblement l'écriture du programme. Dans notre implémentation, des routines sont activées, désactivées, réactivées... Dans la plupart des cas, elles doivent pouvoir se rappeler leur état précédent. Ceci est très facile à réaliser en Fortran grâce au "common".

Un autre avantage non négligeable du Fortran standard est son adaptabilité sur toutes les machines. Il permet également une lecture aisée pour le néophyte.

2) Langage pour l'interpréteur :

Vu le caractère récursif de l'évaluateur, il était souhaitable d'utiliser un langage récursif : - Fortran (sur Burroughs)
- Algol (sinon)

L'évaluateur a été définitivement mis au point sur le Siemens, d'où le choix de l'Algol.

Remarque :

Un langage permettant la récursivité ne s'imposait pas. Il a été envisagé de l'écrire en Fortran, en simulant la récursivité. Cette idée fut abandonnée car elle a tendance à rendre le programme illisible.

CHAPITRE III :

CONCLUSIONS ET EXTENSIONS POSSIBLES

(implémentation)

§1. Lucid : un langage, mais aussi une nouvelle orientation de l'informatique

Lucid se différencie de tous les langages par son but : la vérification de programmes. La façon d'aborder ce problème est également différente des différentes approches déjà faites dans ce domaine. Les auteurs adoptent ou inventent de nouveaux outils de telle sorte que ceux-ci respectent le but du langage. Ces caractéristiques se sont ressenties dans l'implémentation du langage.

- Les opérateurs "α", "β", "γ"

  Alors que l'implémentation de la structure de bloc, des fonctions, ... est résolue au moyen d'un stack et des pointeurs en Algol, des opérateurs ont été défini formellement pour répondre au problème. Les "pointeurs" ont donc une signification.

- Le programme Lucode :

  Le programme Lucid a été traduit en Lucode : programme également non séquentiel Ceci remet également en question la configuration interne des programmes. Supposons, quelques instants, qu'un programme écrit dans un langage quelconque soit traduisible en quelque chose de similaire du Lucode. Serait-ce encore nécessaire d'avoir une machine exigeant que la configuration interne d'un programme soit séquentielle ?

Enfin, comme il l'a été dit dans l'avant-propos. Lucid remet également en question la façon de résoudre un problème par un informaticien.

§2 Extensions immédiates

- Le transformeur n'accepte pas de nombres réels (ceux-ci sont représentés par le quotient de deux entiers).

  Cette extension ne joue qu'au niveau lexicographique.

- "Tupling"

  Une équation Lucid définit une variable.

  Il est possible d'étendre cette notion à un n-uple

  ex : $(V_1, V_2, V_3 \ldots) = \ldots + (T_1, T_2, T_3 \ldots)$ asa ...

  L'évaluation aura besoin d'un troisième paramètre :

  EVAL(m, k, n)

numéro de la variable à évaluer — temps — numéro de la composante du n-uple à évaluer.

- Problème d'entrée-sortie
  . Il doit être possible de lire des données à l'intérieur d'un programme Lucid. Celles-ci ne seraient lues que lorsque l'évaluateur en a besoin.
  . Il faut pouvoir imprimer la valeur d'une variable, une fois évaluée.

§3 Extension dans un avenir plus lointain

Debaixo de boa palava,
ahi está o engano.

Sous les beaux discours
se cache un mauvais tour.
(proverbe portugais)

- Les matrices

  L'interpréteur du Basic Lucid traitait cette structure de donnée. [1]
  Nous préférons cependant attendre la publication d'une théorie sur les matrices en Lucid.

Bien que la méthode proposée (elle consiste à introduire un paramètre "espace" pour l'évaluateur) soit simple, l'implémentation pose des problèmes dans le cas d'un interpréteur amélioré. Ce dernier mémorise en effet la valeur de variables déjà évaluées. Dans le cas d'appels différents à une même fonction, ayant une matrice comme paramètre, nous aurions plusieurs copies de la matrice (copies qui peuvent se différencier de quelques valeurs seulement). L'interpréteur nécessiterait donc une grande place mémoire. Se pose alors le problème de ce qu'il faut ou ne faut pas mémoriser !

- Il est possible d'implémenter Lucid suivant un système de flux de données. Il serait intéressant partant des deux types d'implémentations : - interpréteur
  - flux de données
  de déduire des caractéristiques de programmes favorisant l'une ou l'autre implémentation et d'utiliser ainsi la méthode optimale suivant le programme.

- Autres structures de données : graphe - pile - liste ...
  Ceci rejoint la façon dont Lucid évolue :
  - ces structures restent-elles indispensables ?
  - Si oui, comment les adapter à Lucid sans modifier l'objectif de Lucid.

REFERENCES

[1] Ashcroft E.A. and Wadge W.W., "*Lucid, a non-procedural language with iteration*".
CACM 20, N° 7, pp. 519-526

[2] Christoph M. Hoffmann, *Compilation of a proof oriented language*
IRISA Université de Rennes - iii(ifip), irisa, afcet, pp. 20.1-20.12

[3] Ashcroft E.A and Wadge W.W., "*Lucid : scope structures and defined functions*"
University of Waterloo - Warwick

[4] Ashcroft E.A and Wadge W.W., "*Lucid - a formal system for writing and proving programs*"
SIAM J. COMPUT.5, N° 3, pp. 336-354

FACULTÉ UNIVERSITAIRES NOTRE-DAME DE LA PAIX
NAMUR
INSTITUT D'INFORMATIQUE

Année académique 1977-1978

UNE IMPLÉMENTATION DE

LUCID

(ANNEXE)

Patrick GARDIN

FACULTÉ UNIVERSITAIRES NOTRE-DAME DE LA PAIX
NAMUR
INSTITUT D'INFORMATIQUE

Année académique 1977-1978

UNE IMPLÉMENTATION DE

LUCID

(ANNEXE)

Patrick GARDIN

```
      PROGRAM TRSDRI
C     --------------
C        REORDER THE LUCODE
C
      COMMON/BUFFER/OP(125),OPDE1(125),OPDE2(125),LUCCAT(20),UNUSED(105)
      COMMON/LUCODE/P(4,125)
      COMMON/INDEX/LL,UL
      COMMON/CLCAT/CLCAT(3,20)
      COMMON/NERR/NERR
      COMMON/FLAG/FLAG
      COMMON/CLPOS/LEVEL,NOB
      COMMON/PARAM/PAR1,PAR2,FLOUT,PAR4,PAR5,PAR6
      INTEGER PAR1,PAR2,FLOUT,PAR4,PAR5,OP,OPDE1,OPDE2,CLCAT,UNUSED,
     1        ERRCOD,UL,HGK,PAR6,SW1,P
      LOGICAL FLAG

      ERRCOD = 1
      CALL ERR(ERRCOD)
      CALL RENDRI

      NOB = 1
      FLAG = .FALSE.
      LUCCAT(1) = 1
      HGK = 0

   10 CALL LIMIT(NOB)

      IF (LL.NE.0) GO TO 20
C
C        NEW CLAUSE
C        ----------
C
      NOB = NOB + 1
      LUCCAT(NOB) = HGK + 1
      IF (UL.EQ.0) GO TO 40
      GO TO 10

   20 I1 = LUCCAT(NOB) - 1
      DO 30 I = LL,UL
      K = P(1,I) + I1
      IF (K.GT.HGK) HGK = K
      OP(K) = P(2,I)
      OPDE1(K) = P(3,I)
   30 OPDE2(K) = P(4,I)
      GO TO 10
```

```
C
C          END OF REORDERING. CONSTRUCT THE INTERFACE
C          ------------------------------------------
C
40      CALL CONTRL
        ERRCOD = 2
        CALL ERR(ERRCOD)
        IF (NERR.GT.0) GO TO 50

        SW1 = 1
        CALL SETSW(5,SW1)

        WRITE(FLOUT,1) (NOB,HGK)
1       FORMAT(1X,2I5,/)
        WRITE(FLOUT,2) (OP(I),I=1,HGK)
2       FORMAT(1X,7I10,/)
        WRITE(FLOUT,2) (OPDE1(I),I=1,HGK)
        WRITE(FLOUT,2) (OPDE2(I),I=1,HGK)
        WRITE(FLOUT,2) (LUCCAT(I),I=1,NOB)
        WRITE(FLOUT,2) ((CLCAT(I,J),J=1,NOB),I=1,3)
50      CONTINUE
        STOP
        END
```

```
      SUBROUTINE LIMIT(N)
C     -------------------
C        GIVES THE LIMITS OF THE DIFFERENT PARTS OF A CLAUSE BODY
C
      COMMON/INDEX/LL,UL
      COMMON/FLAG/FLAG
      COMMON/PICPRG/DSPPIC(2,20),PICPRG(3,40),J
      INTEGER UL,DSPPIC,PICPRG
      LOGICAL FLAG

      IF (FLAG) GO TO 10

C
C        FIRST PART OF A CLAUSE BODY
C        ---------------------------
C
      FLAG = .TRUE.
      J = DSPPIC(1,N)
      LL = PICPRG(1,J)
      UL = PICPRG(2,J)
      IF (UL-LL.LT.0) GO TO 10
      RETURN

C
C        GIVE THE CONTINUATION OF THE CLAUSE BODY
C        ----------------------------------------
C
10    J = PICPRG(3,J)
      IF (J.EQ.0) GO TO 20
      LL = PICPRG(1,J)
      UL = PICPRG(2,J)
      IF (UL-LL.GE.0) RETURN
      GO TO 10

C
C        END OF CLAUSE HAS BEEN REACHED
C        ------------------------------
20    FLAG = .FALSE.
      LL = 0
      UL = 999

      IF (.NOT.(DSPPIC(1,N+1).EQ.999.AND.DSPPIC(2,N+1).EQ.999)) RETURN
C
C        END OF PROGRAM HAS BEEN REACHED
C        -------------------------------
C
      UL = 0
      RETURN
      END
```

```
      SUBROUTINE RENDRI
C     -----------------
C        DRIVER FOR THE VARIABLE RENAMING
C
      COMMON/LIMITS/MAX0,MAX1,MAX2,MAX3,MAX4,MAX5,MAX6,MAX7,MAX8,
     1              MAX9,MAXLUC,MAX11
      COMMON/BUFFER/P(500)
      COMMON/LUCODE/Q(4,125)
      COMMON/INDEX/L,M
      COMMON/LSTNOB/LSTNOB(5)
      COMMON/CLPOS/LEVEL,NOB
      INTEGER P,Q,ERRCOD

      CALL BETA

      LEVEL = 1
      NOB = 1
      LSTNOB(1) = 1
      M = 1
      L = 1
      I = 1

      CALL RENCAL

20    IF (P(L).GT.63) GO TO 30
      IF (P(L).EQ.53) GO TO 40
      IF (P(L).EQ.0.AND.P(L+1).EQ.0) GO TO 50
      IF (P(L).EQ.12.OR.P(L).EQ.16.OR.P(L).EQ.22.OR.P(L).EQ.25)
     1                          GO TO 60
      IF (P(L).EQ.13) GO TO 70
C
C        OPERATOR
C        --------
      Q(I,M) = P(L)
      I = I + 1
      L = L + 1
      GO TO 20
C
C        VARIABLE
C        --------
C
30    CALL RENVR(LEVEL,I)
      I = I + 1
      GO TO 20
```

```
C
C          GO ON WITH NEXT EQUATION
C          ------------------------
C
40      M = M + 1
        IF (M.GT.MAXLUC) GO TO 80
        I = 1
        L = L + 1
        GO TO 20
C
C          ALREADY RENAMED BY RENCAL
C          -------------------------
C
50      L = L + 5
        GO TO 20
C.
C          CLAUSE HEAD
C          -----------
C
60      CALL RENHD
        I = 1
        GO TO 20
C
C          END OF CLAUSE
C          -------------
C
70      CALL ENDOCL
        I = 1
        IF (P(L).EQ.54) RETURN
        GO TO 20
C
C          TOO MANY LUCODE EQUATIONS
C
80      ERRCOD = 30
        CALL ERR(ERRCOD)
.       RETURN
        END
```

```
      SUBROUTINE RENCAL
C     -----------------
C        RENAMES THE CLAUSE CALLS
C
      COMMON/LIMITS/MAX0,MAX1,MAX2,MAX3,MAX4,MAX5,MAX6,MAX7,MAX8,
     1              MAX9,MAXLUC,MAX11
      COMMON/BUFFER/P(500)
      COMMON/LUCODE/Q(4,125)
      COMMON/INDEX/L,M
      COMMON/CLPOS/LEVEL,NOB
      COMMON/CALADD/STADD(2,50),IADD,DSPADD(2,20)
      COMMON/CLCAT/CLCAT(3,20)
      INTEGER P,Q,STADD,DSPADD,CLCAT,ERRCOD

      LSTM = M
      CLCAT(2,NOB) = 0
      J = DSPADD(1,NOB)
      IF (J.EQ.0) RETURN
C
C        RENAMES THE LEFT HAND SIDE OF AN EQUATION
C        -----------------------------------------
C

      I2 = 1
10    I = STADD(1,J)
      CALL ADELE(P(I),NEW,I2,LEVEL)
      Q(1,M) = NEW
      DO 20 K = 2,4
      I1 = I + K - 1
20    Q(K,M) = P(I1)

      P(I) = 0
      P(I+1) = 0
      M = M + 1
      IF (M.GT.MAXLUC) GO TO 50
      J = STADD(2,J)
      IF (J.EQ.0) GO TO 30
      GO TO 10
C
C        RENAMES THE RIGHT HAND SIDE OF AN EQUATION
C        ------------------------------------------
C

30    CLCAT(2,NOB) = Q(1,LSTM)
      I1 = M - 1
      DO 40 K = LSTM,I1
      DO 40 KK = 3,4
      CALL ADELE(Q(KK,K),NEW,KK,LEVEL)
40    Q(KK,K) = NEW

```

```
      RETURN
C
C        TOO MANY EQUATIONS
C
50    ERRCOD = 30
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE RENVR(LEVEL,I)
C     -------------------------
C        RENAMES VARIABLES
C
      COMMON/LUCODE/Q(4,125)
      COMMON/BUFFER/P(500)
      COMMON/INDEX/L,M
      INTEGER P,Q

      J = LEVEL
      IF (L.LE.2) GO TO 10
C
C        FOR GLOBALS TAKE THE LEVEL WHERE THEY WERE DEFINED
C        --------------------------------------------------
C
      IF (P(L-2).EQ.3) J = LEVEL - P(L-1) + 1000
10    CALL ADELE(P(L),NEW,I,J)
      Q(I,M) = NEW
      L = L + 1
      RETURN
      END
```

```
      SUBROUTINE RENHD
C     ----------------
C        SUPRESS THE CLAUSE HEAD
C
      COMMON/CLPOS/LEVEL,NOB
      COMMON/INDEX/L,M

      CALL ADGLB
      CALL ADNAM(L)
      CALL PHBGOB(M)
      CALL RENCAL
      L = L + 3
      RETURN
      END
```

```
      SUBROUTINE ENDOCL
C     ----------------
C        SUPRESS THE END OF A CLAUSE
C
      COMMON/INDEX/L,M
      COMMON/CLPOS/LEVEL,NOB
      COMMON/REN/RENTAB(2,100),IREN,DSPREN(5),NEWNAM(100)
      INTEGER RENTAB,DSPREN

      I1 = DSPREN(LEVEL)
      CALL BADDEF(LEVEL)
      CALL PHEOB(M)
      IREN = I1
      L = L + 2
      RETURN
      END
```

```
      SUBROUTINE ADELE(NAME,J,I,LEVEL)
C     -------------------------------
C        ADD THY ELEMENT "NAME" IN NEWNAM IF NOT IN IT
C
      COMMON/LIMITS/MAX0,MAXIDN,MAX2,MAX3,MAX4,MAX5,MAX6,MAX7,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/REN/RENTAB(2,100),IREN,DSPREN(5),NEWNAM(100)
      INTEGER RENTAB,DSPREN,SEARCH,ERRCOD

      IF (NAME.GE.500) GO TO 30

      J = SEARCH(NAME,DSPREN(LEVEL),IREN,NEWNAM)
      IF (J.GT.0) GO TO 10

      NEWNAM(IREN) = NAME
      RENTAB(1,IREN+1) = 0
      RENTAB(2,IREN+1) = 0
      J = IREN
      IREN = IREN + 1
      IF (IREN.GT.MAXIDN) GO TO 40
10    I1 = J
      J = J - DSPREN(LEVEL) + 1
      IF (I.EQ.1) GO TO 20

      RENTAB(2,I1) = 1
      RETURN

20    RENTAB(1,I1) = RENTAB(1,I1) + 1
      RETURN

30    J = NAME
      RETURN
C
C        NEWNAM IS FULL
C
40    ERRCOD = 7
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE ADNAM(L)
C     -------------------
C        PUT THY GOOD "OUTPUT" NAME OF A CLAUSE IN NEWNAM
C
      COMMON/LIMITS/MAX0,MAXIDN,MAX2,MAX3,MAX4,MAX5,MAX6,MAX7,MAX8,
     1             MAX9,MAX10,MAx11
      COMMON/BUFFER/P(500)
      COMMON/REN/RENTAB(2,100),IREN,DSPREN(5),NEWNAM(100)
      COMMON/CLPOS/LEVEL,NOB
      INTEGER P,RENAMT,DSPREN,ERRCOD

      DSPREN(LEVEL+1) = IREN
      RENTAB(1,IREN) = 0
      RENTAB(2,IREN) = 0
C
C        RESULT ("65") FOR PRO-FUN AND OUTPUT ("64") FOR CMP-MAP
C        -------------------------------------------------------
C
      NEWNAM(IREN) = 64
      IF (P(L).EQ.25.OR.P(L).EQ.16) NEWNAM(IREN) = 65
      IREN = IREN + 1
      IF (IREN.GT.MAXIDN) GO TO 20
      RENTAB(1,IREN) = 0
      RENTAB(2,IREN) = 0
      RETURN

C
C        NEWNAM IS FULL
C
20    ERRCOD = 7
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE ADGLB
C     ----------------
C        RENAME THE GLOBALS (GLOBALS FOR ONLY ONE LEVEL)
C
      COMMON/CLPOS/LEVEL,NOB
      COMMON/GLB/GLBNST(2,80),GLBTAB(80),IGLB,DSPGLB(20)
      INTEGER GLBNST,GLBTAB,DSPGLB

      I3 = 2
      I1 = NOB + 2
      I2 = DSPGLB(NOB+1)
      IF (I2.EQ.DSPGLB(I1)) RETURN
      I1 = DSPGLB(I1) - 1
      DO 10 K = I2,I1
      IF (GLBNST(1,K).NE.1) GO TO 10

      CALL ADELE(GLBTAB(K),NEW,I3,LEVEL)
10    CONTINUE

      RETURN
      END
```

```
      SUBROUTINE PHBGOB(M)
C     --------------------
C        TAKES A PICTURE OF WHAT IS STILL DONE AT THE BEGIN OF A BLOCK
C
      COMMON/LIMITS/MAX0,MAX1,MAX2,MAX3,MAX4,MAX5,MAX6,MAX7,MAX8,
     1              MAX9,MAX10,MAXPIC
      COMMON/PICPRG/DSPPIC(2,20),PICPRG(3,40),IPIC
      COMMON/CLPOS/LEVEL,NOB
      COMMON/LSTNOB/LSTNOB(5)
      INTEGER DSPPIC,PICPRG,ERRCOD

      PICPRG(2,IPIC) = M - 1
      I1 = LSTNOB(LEVEL)
      DSPPIC(2,I1) = IPIC
      NOB = NOB + 1
      LEVEL = LEVEL + 1
      LSTNOB(LEVEL) = NOB
      IPIC = IPIC + 1
      IF (IPIC.GT.MAXPIC) GO TO 10
      DSPPIC(1,NOB) = IPIC
      PICPRG(1,IPIC) = M
      RETURN
C
C        THE ROLL-FILM IS TOO SMALL
C
10    ERRCOD = 31
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE PHEOB(M)
C     -------------------
C        TAKES A PICTURE OF WHAT IS ALREADY DONE AT THE END OF A CLAUSE
C
      COMMON/LIMITS/MAX0,MAX1,MAX2,MAX3,MAX4,MAX5,MAX6,MAX7,MAX8,
     1             MAX9,MAX10,MAXPIC
      COMMON/PICPRG/DSPPIC(2,20),PICPRG(3,40),IPIC
      COMMON/LSTNOB/LSTNOB(5)
      COMMON/CLPOS/LEVEL,NOB
      INTEGER DSPPIC,PICPRG,ERRCOD

      PICPRG(2,IPIC) = M - 1
      I1 = LSTNOB(LEVEL)
      DSPPIC(2,I1) = IPIC
      LEVEL = LEVEL - 1
      IPIC = IPIC + 1
      IF (IPIC.GT.MAXPIC) GO TO 20
      IF (LEVEL.EQ.0) GO TO 10
      I1 = LSTNOB(LEVEL)
      I2 = DSPPIC(2,I1)
      PICPRG(3,I2) = IPIC
      PICPRG(1,IPIC) = M
      RETURN

C
C        END OF FILE
C        -----------
C
10    DSPPIC(1,NOB+1) = 999
      DSPPIC(2,NOB+1) = 999
      RETURN
C
C        THE ROLL-FILM IS TOO SMALL
C
20    ERRCOD = 31
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE BETA
C     ---------------
C        BETA IDTRODUCER DRIVER
C
      COMMON/LUCODE/P(500)
      COMMON/LSTTYP/LSTTYP(5)
      COMMON/INDEX/L,M
      COMMON/LSTNOB/LSTNOB(5)
      COMMON/CLPOS/LEVEL,NOB
      INTEGER P

      CALL SYNAN

      M = 0
      L = 1
      LEVEL = 1
      NOB = 1
      LSTNOB(1) = 1
      LSTTYP(1) = 25

10    IF (P(L).GT.63) GO TO 20
      IF (P(L).EQ.13) GO TO 30
      IF (P(L).EQ.54) GO TO 40

      CALL CLHEAD
      GO TO 10

20    CALL EQUATN
      GO TO 10

30    CALL ENDCL
      GO TO 10

40    LENGHT = 2
      CALL COPY(LENGHT)
      RETURN
      END
```

```
      SUBROUTINE CLHEAD
C     -----------------
C        RENAMES THE CLAUSE NAMES
C
C
      COMMON/INDEX/L,M
      COMMON/LUCODE/P(500)
      COMMON/BUFFER/Q(500)
      COMMON/LSTNOB/LSTNOB(5)
      COMMON/CLPOS/LEVEL,NOB
      COMMON/CALADD/STADD(2,50),IADD,DSPADD(2,20)
      COMMON/LSTTYP/LSTTYP(5)
      COMMON/CLCAT/CLCAT(3,20)
      INTEGER P,Q,STADD,DSPADD,CLCALL,CLCAT

      LEVEL = LEVEL + 1
      LSTTYP(LEVEL) = P(L)
      LENGHT = 3
      CALL COPY(LENGHT)
      I1 = P(L-1)
      I2 = LEVEL - 1
      Q(M-1) = CLCALL(I1,I2) + 500
      DSPADD(2,NOB) = IADD
      I2 = IADD + 1
      IF (DSPADD(1,NOB).EQ.I2) DSPADD(1,NOB) = 0
      NOB = NOB + 1
      CLCAT(3,NOB) = 0
      DSPADD(1,NOB) = IADD + 1
      DSPADD(2,NOB) = 0
      LSTNOB(LEVEL) = NOB
      L = L + 1
      RETURN
      END
```

```
      SUBROUTINE ENDCL
C     ----------------
C        UPDATE OF POINTERS IN DSPADD
C
      COMMON/LSTNOB/LSTNOB(5)
      COMMON/CLPOS/LEVEL,NOB
      COMMON/CALADD/STADD(2,50),IADD,DSPADD(2,20)
      COMMON/INDEX/L,M
      INTEGER STADD,DSPADD

      I1 = LSTNOB(LEVEL)
      IF (DSPADD(2,I1).EQ.0) DSPADD(1,I1) = 0
      LEVEL = LEVEL - 1
      I1 = LSTNOB(LEVEL)
      IF (DSPADD(1,I1).EQ.0) DSPADD(1,I1) = IADD + 1
      LENGHT = 2
      CALL COPY(LENGHT)
      L = L + 1
      RETURN
      END
```

```
      SUBROUTINE EQUATN
C     -----------------
C        THE INTRODUCING OF BETA DEPENDS OF THE KIND OF EQUATION
C
      COMMON/LUCODE/P(500)
      COMMON/BUFFER/Q(500)
      COMMON/INDEX/L,M
      COMMON/CLPOS/LEVEL,NOB
      INTEGER P,Q,CLCALL,ERRCOD

      ERRCOD = 29
      IF (CLCALL(P(L),LEVEL).NE.0) CALL ERR(ERRCOD)
      I2 = L + 1
      IF (P(I2).EQ.3) GO TO 40
      IF (P(I2).EQ.56) GO TO 30
C
C        USUAL EQUATION: VAR = OPERATOR OPERAND OPERAND
C        ----------------------------------------------
      LENGHT = 4
      IF (P(L+4).EQ.53) LENGHT = 5
      IF (LENGHT-4) 20,20,10

10    I2 = P(L+3)
      CALL TSTLOC(I2,LEVEL)
20    I2 = P(L+2)
      CALL TSTLOC(I2,LEVEL)
      CALL COPY(LENGHT)
      L = L + 1
      RETURN
C
C        EQUATION WITH A "CLAUSE APPLICATION" AS OPERATOR
C        ------------------------------------------------

30    P(I2) = 2
      CALL FUNMAP
      L = L + 1
      RETURN
C
C        EQUATION IN GAMMA
C        -----------------

40    CALL TSTGLB(LEVEL)
      LENGHT = 5
      CALL COPY(LENGHT)
      L = L + 1
      RETURN
      END
```

```
      SUBROUTINE COPY(LENGHT)
C     ----------------------
C        STANDARD ROUTINE
C
      COMMON/LIMITS/MAX0,MAX1,MAX2,MAX3,MAX4,MAX5,MAXSYN,MAX7,MAX8,
     1             MAX9,MAX10,MAx11
      COMMON/LUCODE/P(500)
      COMMON/BUFFER/Q(500)
      COMMON/INDEX/L,M
      INTEGER P,Q,ERRCOD

      M = M + LENGHT
      IF (M.GT.MAXSYN) GO TO 20
      L = L + LENGHT - 1
      DO 10 K = 1,LENGHT
      I1 = M - K + 1
      I2 = L - K + 1
10    Q(I1) = P(I2)

      RETURN
C
C        Q IS FULL
C
20    ERRCOD = 22
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE TSTLOC(NAME,LEVEL)
C     ------------------------------
C        ANALYSE THE POSSIBILITY FOR INTRODUCING "BETA" IN A USUAL
C        EQUATION
C
      COMMON/LUCODE/P(500)
      COMMON/INDEX/L,M
      INTEGER P,CLCALL

      I = CLCALL(NAME,LEVEL)
      IF (I.EQ.0) RETURN

      CALL LOCCL(I,NAME)
      RETURN
      END
```

```
      SUBROUTINE TSTGLB(LEVEL)
C     ------------------------
C        ANALYSE THE POSSIBILITY FOR INTRODUCING "BETA" IN A GAMMA
C        EQUATION
C
      COMMON/BU1FER/Q(500)
      COMMON/LUCODE/P(500)
      COMMON/INDEX/L,M
      INTEGER P,Q,CLCALL

      I1 = LEVEL + 1000 - P(L+2)
      I2 = P(L+3)
      I = CLCALL(I2,I1)
      IF (I.EQ.0) RETURN

      CALL GLBCL(I,I2)
      RETURN
      END
```

```
      SUBROUTINE FUNMAP
C     -----------------
C        ANALYSE A CLAUSE CALL EQUATION (CLAUSE WITH ARGUMENTS)
C
      COMMON/BUFFER/Q(500)
      COMMON/LUCODE/P(500)
      COMMON/INDEX/L,M
      COMMON/CLCAT/CLCAT(3,20)
      COMMON/CLPOS/LEVEL,NOB
      COMMON/LSTNOB/LSTNOB(5)
      INTEGER P,Q,CLCAT,CLCALL

      CALL UPDCAL(M)
      LENGHT = 5
      CALL COPY(LENGHT)
      I1 = LSTNOB(LEVEL)
      CLCAT(3,I1) = CLCAT(3,I1) + 1
      I = CLCALL(P(L-2),LEVEL)
      IF (I.NE.0) Q(M-2) = I + 500
      RETURN
      END
```

```
      SUBROUTINE UPDCAL(M)
C     --------------------
C        REMEMBER THE DIFFERENT CLAUSE CALLS
C
      COMMON/LIMITS/MAX0,MAX1,MAX2,MAX3,MAX4,MAX5,MAX6,MAX7,MAX8,
     1              MAXCAL,MAX10,MAX11
      COMMON/CALADD/STADD(2,50),IADD,DSPADD(2,20)
      COMMON/LSTNOB/LSTNOB(5)
      COMMON/CLPOS/LEVEL,NOB
      INTEGER STADD,DSPADD,ERRCOD

      I1 = LSTNOB(LEVEL)
      I2 = DSPADD(2,I1)
      IF (I2.EQ.0) GO TO 10

      STADD(2,I2) = IADD + 1

10    IADD = IADD + 1
      IF (IADD.GT.MAXCAL) GO TO 20
      STADD(1,IADD) = M + 1
      DSPADD(2,I1) = IADD
      RETURN
C
C        STADD IS FULL
C
20    ERRCOD = 27
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE LOCCL(I,NAME)
C     ------------------------
C        INTRODUCE "BETA" FOR A LOCAL CLAUSE CALL
C
      COMMON/LIMITS/MAX0,MAX1,MAX2,MAX3,MAX4,MAX5,MAXSYN,MAX7,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/BUFFER/Q(500)
      COMMON/LUCODE/P(500)
      COMMON/INDEX/L,M
      COMMON/LSTNOB/LSTNOB(5)
      COMMON/CLPOS/LEVEL,NOB
      COMMON/CLCAT/CLCAT(3,20)
      INTEGER P,Q,CLCAT,ERRCOD

      IF (P(L+1).EQ.26) GO TO 10
      I2 = LSTNOB(LEVEL)

5     CALL UPDCAL(M)
      M = M + 5
      IF (M.GT.MAXSYN) GO TO 20
      Q(M) = 53
      Q(M-1) = 700
      Q(M-2) = I + 500
      Q(M-3) = 2
      Q(M-4) = NAME
      CLCAT(3,I2) = CLCAT(3,I2) + 1
      RETURN
C
C        CLAUSE NAME AS PARAMETER
C        ------------------------
C

10    ERRCOD = 28                                          go to 5
      IF (~~~~(CLCAT(1,I).EQ.12.OR.CLCAT(1,I).EQ.25)) ~~CALL ERR(ERRCOD)~~
      P(L+2) = I + 500   < call err (errcod)
      RETURN
C
C        Q IS FULL
C
20    ERRCOD = 22
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE GLBCL(I,NAME)
C     ------------------------
C        INTRODUCE "BETA" FOR A GLOBAL CLAUSE CALL
C
      COMMON/LIMITS/MAX0,MAX1,MAX2,MAX3,MAX4,MAX5,MAXSYN,MAX7,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/BUFFER/Q(500)
      COMMON/LUCODE/P(500)
      COMMON/INDEX/L,M
      COMMON/CLCAT/CLCAT(3,20)
      COMMON/CLPOS/LEVEL,NOB
      COMMON/LSTTYP/LSTTYP(5)
      INTEGER P,Q,ERRCOD,CLCAT

      IF (LSTTYP(LEVEL).EQ.12.OR.LSTTYP(LEVEL).EQ.22) CALL NIL(I)
      IF (CLCAT(1,I).EQ.16.OR.CLCAT(1,I).EQ.22) GO TO 10

      CALL UPDCAL(M)
      M = M + 5
      IF (M.GT.MAXSYN) GO TO 20
      Q(M) = 53
      Q(M-1) = 700
      Q(M-2) = NAME
      Q(M-3) = 2
      Q(M-4) = P(L)
      P(L) = NAME
      P(L+3) = I + 500
      RETURN

C
C        GLB FUN OR MAP : "BETA" WAS ALREADY INTRODUCED
C        ----------------------------------------------

10    P(L+3) = I + 500
      RETURN
C
C        Q IS FULL
C
20    ERRCOD = 22
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE NIL(I)
C     ----------------
C        NIL = NONE ITERATIVE LANGUAGE
C        LOOCK IF THERE ARE COMON GLOBALS
C
      COMMON/GLB/GLBNST(2,80),GLBTAB(80),IGLB,DSPGLB(20)
      COMMON/LSTNOB/LSTNOB(5)
      COMMON/CLPOS/LEVEL,NOB
      INTEGER GLBNST,GLBTAB,DSPGLB,ERRCOD,SEARCH,ULI,ULK

      ERRCOD = 40
      K = LSTNOB(LEVEL)
      LLK = DSPGLB(K)
      ULK = DSPGLB(K+1)
      LLI = DSPGLB(I)
      ULI = DSPGLB(I+1)
      IF (ULI.EQ.LLI) RETURN
      ULI = ULI - 1
      DO 10 K = LLI,ULI
      I1 = GLBTAB(K)
      IF (SEARCH(I1,LLK,ULK,GLBTAB).GT.0) CALL ERR(ERRCOD)
10    CONTINUE
      RETURN
      END
```

```
      SUBROUTINE SYNAN
C     ----------------
C        BRINGS THE PROGRAM TOGETHER
C
      COMMON/LIMITS/MAX0,MAX1,MAX2,MAX3,MAX4,MAX5,MAXSYN,MAX7,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/BUFFER/P(216),UNUSED(284)
      COMMON/LUCODE/Q(500)
      COMMON/INDEX/L,M
      COMMON/CLPOS/LEVEL,NOB
      INTEGER P,Q,UNUSED,ERRCOD

      MM = 0
10    CALL OPDRI
      M = MM
      IF (P(1).EQ.54) GO TO 20
      CALL TYPE
      MM = M
      GO TO 10
C
C        END OF FILE
C        -----------
20    M = M + 1
      IF (M.GT.MAXSYN) GO TO 30
      Q(M) = 54
      ERRCOD = 9
      IF (LEVEL.NE.0) CALL ERR(ERRCOD)
      RETURN
C
C        Q IS FULL
C
30    ERRCOD = 22
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE TYPE
C     ---------------
C        ASK RGHDSD FOR AN EXPRESSION ANALYSING
C        INTRODUCE LAMBDA FOR FORMAL PARAMETERS
C
      COMMON/LIMITS/MAX0,MAXIDN,MAX2,MAX3,MAX4,MAX5,MAXSYN,MAX7,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/BUFFER/P(216),UNUSED(284)
      COMMON/LUCODE/Q(500)
      COMMON/INDEX/L,M
      COMMON/IDN/TEMPVR,IDNTAB(100)
      INTEGER P,Q,UNUSED,TEMPVR,ERRCOD

      IF ((P(1).GT.63.AND.P(1).LT.1000).AND.P(2).EQ.55) GO TO 10
      IF (P(1).EQ.12.OR.P(1).EQ.16.OR.P(1).EQ.22.OR.P(1).EQ.25.OR.
     1            P(1).EQ.13) GO TO 20
      ERRCOD = 21
      CALL ERR(ERRCOD)
      RETURN
C
C        EXPRESSION ANALYSING
C        --------------------

10    L = 3
      CALL RGHDSD
      RETURN

20    LENGHT = 2
      IF (P(2).NE.53) LENGHT = 3
      M = M + LENGHT
      IF (M.GT.MAXSYN) GO TO 60
      DO 30 K = 1,LENGHT
      I1 = M - K + 1
      I2 = LENGHT - K + 1
30    Q(I1) = P(I2)
      IF (Q(M).EQ.53) RETURN
      Q(M) = 53
C
C        INTRODUCING OF ALPHA
C        --------------------

      L = 3
40    M = M + 4
      IF (M.GT.MAXSYN) GO TO 60
      Q(M) = 53
      Q(M-1) = L - 2 + 1000
      Q(M-2) = 1
      Q(M-3) = P(L)
      IF (P(1).NE.22) GO TO 50

```

```
51
52 C
53 C          INTRODUCING OF LATEST
54 C          ---------------------
55          TEMPVR = TEMPVR + 1
56          IF (TEMPVR.GT.MAXIDN) GO TO 70
57          Q(M-3) = TEMPVR
58          M = M + 5
59          IF (M.GT.MAXSYN) GO TO 60
60          Q(M) = 53
61          Q(M-1) = TEMPVR
62          Q(M-2) = 1001
63          Q(M-3) = 4
64          Q(M-4) = P(L)
65 50       L = L + 1
66          IF (P(L).EQ.53) RETURN
67          GO TO 40
68 C
69 C          Q IS FULL
70 C
71 60       ERRCOD = 22
72          CALL ERR(ERRCOD)
73          RETURN
74 C
75 C          NO MORE SPACE IN IDNTAB
76 C
77 70       ERRCOD = 7
78          CALL ERR(ERRCOD)
79          RETURN
80          END
```

```
      SUBROUTINE RGHDSD
C     -----------------
C        HANDLE THE RIGHT HAND SIDE OF AN EQUATION
C
      COMMON/LIMITS/MAX0,MAX1,MAX2,MAX3,MAX4,MAX5,MAXSYN,MAX7,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/BUFFER/P(216),UNUSED(284)
      COMMON/LUCODE/Q(500)
      COMMON/INDEX/L,M
      COMMON/IDN/TEMPVR,IDNTAB(100)
      INTEGER P,Q,UNUSED,ARIT,ERRCOD,TEMPVR

      IF (P(L+1).EQ.53) GO TO 10

      CALL EXP(ARIT)
      I1 = M - (ARIT+2)
      Q(I1) = P(1)
      TEMPVR = TEMPVR - 1
      RETURN

C
C        LUCODE EQUATION DIRECTLY PRODUCED
C        ---------------------------------
10    L = L + 1
      M = M + 4
      IF (M.GT.MAXSYN) GO TO 30
      I1 = M - 3
      DO 20 K = I1,M
      Q(K) = P(L-3)
20    L = L + 1
      RETURN
C
C        Q IS FULL
C
30    ERRCOD = 22
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE EXP(ARIT)
C     --------------------
C        EXPRESSION ANALYSER
C
      COMMON/LIMITS/MAX0,MAX1,MAX2,MAX3,MAX4,MAX5,MAX6,MAXVR,MAXOP,
     1              MAX9,MAX10,MAX11
      COMMON/BUFFER/P(216),UNUSED(284)
      COMMON/LUCODE/Q(500)
      COMMON/INDEX/L,M
      COMMON/PREC/PREC(64)
      COMMON/ST/VRST(50),IVR,OPST(25),IOP
      COMMON/FLAG/FLAG
      INTEGER P,Q,UNUSED,VRST,OPST,OPCODE,TOKEN,PREC,ARIT,ERRCOD
      LOGICAL FLAG

      FLAG = .TRUE.
      IVR = 1
      IOP = 1
      OPST(1) = 11

10    TOKEN = P(L)
      IF (TOKEN.LE.63) GO TO 20
      VRST(IVR) = TOKEN
      IVR = IVR + 1
      IF (IVR.GT.MAXVR) GO TO 40
      L = L + 1
      GO TO 10

20    OPCODE = OPST(IOP)
      IF (PREC(OPCODE+1).LE.PREC(TOKEN+1).OR.TOKEN.EQ.42) GO TO 30
      IF (OPCODE.EQ.11) RETURN
      CALL CODGEN(ARIT,TOKEN)
      GO TO 10

30    IOP = IOP + 1
      IF (IOP.GT.MAXOP) GO TO 50
      OPST(IOP) = TOKEN
      L = L + 1
      GO TO 10

C
C        NO MORE SPACE IN THE OPERAND STACK
C
40    ERRCOD = 23
      CALL ERR(ERRCOD)
      RETURN




C
C        NO MORE SPACE IN THE OPERATOR STACK
C
50    ERRCOD = 24
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE CODGEN(ARIT,TOKEN)
C     ----------------------------
C        LUCODE GENERATOR
C
      COMMON/LUCODE/Q(500)
      COMMON/INDEX/L,M
      COMMON/ARITY/ARITY(64)
      COMMON/ST/VRST(50),IVR,OPST(25),IOP
      COMMON/FLAG/FLAG
      INTEGER Q,ARITY,VRST,OPST,OPCODE,ARIT,TOKEN,ERRCOD
      LOGICAL FLAG

      IF (OPST(IOP).EQ.44.OR.((OPST(IOP).EQ.42.AND.OPST(IOP-1).EQ.56)
     1             .AND.TOKEN.EQ.43.AND.FLAG)) GO TO 40

10    OPCODE = OPST(IOP)
      IOP = IOP - 1
      IF (OPCODE.EQ.42.OR.OPCODE.EQ.18) GO TO 30

20    IF (IVR.LE.ARITY(OPCODE+1)) GO TO 60
      IF (ARITY(OPCODE+1).LT.0) GO TO 70
      IVR = IVR - ARITY(OPCODE+1)
      CALL EMIT(OPCODE)
      ARIT = ARITY(OPCODE+1)
      RETURN
C
C        GO FUTHER WITH ANALYSING
C        ------------------------
30    L = L + 1
      FLAG = .TRUE.
      RETURN

40    CALL COMMA
      FLAG = .FALSE.
      GO TO 10

C
C        NO OPERANDS ENOUGH
C
60    ERRCOD = 25
      CALL ERR(ERRCOD)
      RETURN
C
C        INVALID OPERATOR
C
70    ERRCOD = 26
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE EMIT(OPCODE)
C     ----------------------
C        PUTS THE LUCODE OUT
C
      COMMON/LIMITS/MAX0,MAXIDN,MAX2,MAX3,MAX4,MAX5,MAXSYN,MAXVR,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/LUCODE/Q(500)
      COMMON/INDEX/L,M
      COMMON/ST/VRST(50),IVR,OPST(25),IOP
      COMMON/ARITY/ARITY(64)
      COMMON/IDN/TEMPVR,IDNTAB(100)
      INTEGER Q,OPCODE,VRST,OPST,TEMPVR,ARITY,ERRCOD

      TEMPVR = TEMPVR + 1
      IF (TEMPVR.GT.MAXIDN) GO TO 40
      IDNTAB(TEMPVR) = TEMPVR + 63
      I1 = M
      M = M + ARITY(OPCODE+1) + 3
      IF (M.GT.MAXSYN) GO TO 50
      IF (ARITY(OPCODE+1)-1) 30,20,10

10    Q(I1+4) = VRST(IVR+1)
20    Q(I1+3) = VRST(IVR)
30    Q(I1+2) = OPCODE
      Q(I1+1) = TEMPVR + 63
      Q(M) = 53
      VRST(IVR) = TEMPVR + 63
      IVR = IVR + 1
      IF (IVR.GT.MAXVR) GO TO 60
      RETURN
C
C        NO MORE SPACE IN IDNTAB
C
40    ERRCOD = K
      CALL ERR(ERRCOD)
      RETURN
C
C        Q IS FULL
C
50    ERRCOD = 22
      CALL ERR(ERRCOD)
      RETURN
C
C        NO MORE SPACE IN THE OPERAND STACK
C
60    ERRCOD = G3
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE COMMA
C     ----------------
C        HANDLE N-UPLES
C
      COMMON/LIMITS/MAX0,MAX1,MAX2,MAX3,MAX4,MAX5,MAX6,MAXVR,MAXOP,
     1              MAX9,MAX10,MAX11
      COMMON/ST/VRST(50),IVR,OPST(25),IOP
      INTEGER OPST,VRST,ERRCOD

      VRST(IVR) = 700
      IVR = IVR + 1
      IF (IVR.GT.MAXVR) GO TO 40
      K = IOP

10    IF (OPST(K).NE.44) GO TO 20
      OPST(K) = 26
      K = K - 1
      GO TO 10

20    IOP = IOP + 1
      IF (IOP.GT.MAXOP) GO TO 30
      OPST(IOP) = 26
      RETURN
C
C        NO MORE SPACE IN THE OPERATOR STACK
C
30    ERRCOD = 24
      CALL ERR(ERRCOD)
      RETURN
C
C        NO MORE SPACE IN THE VARIABLE STACK
C
40    ERRCOD = 23
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE OPDRI
C     ----------------
C        ANALYSE THE PROBLEM OF GLOBALS
C
      COMMON/LIMITS/MAXLEX,MAX1,MAX2,MAX3,MAX4,MAX5,MAX6,MAX7,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/BUFFER/Q(216),P(216),UNUSED(68)
      COMMON/INDEX/L,M
      COMMON/GLB/GLBNST(2,80),GLBTAB(80),IGLB,DSPGLB(20)
      COMMON/CLPOS/LEVEL,NOB
      COMMON/LSTNOB/LSTNOB(5)
      INTEGER GLBNST,GLBTAB,DSPGLB,SEARCH,ERRCOD,P,Q,UNUSED

      CALL LEXDRI

      ERRCOD = 8
      I1 = LSTNOB(LEVEL)
      M = 0
      L = 1
      IF (P(L).EQ.12.OR.P(L).EQ.16.OR.P(L).EQ.22.OR.P(L).EQ.25) GO TO 30
      IF (P(1).EQ.54) GO TO 50
      IF (SEARCH(P(L),DSPGLB(I1),DSPGLB(I1+1),GLBTAB).GT.0)
     1          CALL ERR(ERRCOD)
10    IF (P(L).GT.63.AND.P(L).LT.1000) GO TO 20
C
C        SEPARATOR
C        ---------

      M = M + 1
      IF (M.GT.MAXLEX) GO TO 60
      Q(M) = P(L)
      IF (P(L).EQ.53) GO TO 40

      L = L + 1
      GO TO 10

C
C        VARIABLE
C        --------

20    CALL GAMMA
      L = L + 1
      GO TO 10
C
C        CLAUSE HEAD
C        -----------

30    CALL CLAUSE
      RETURN

```

```
51 C
52 C          END OF SENTENCE
53 C          ---------------
54 40      IF (P(L-1).NE.13) RETURN
55
56 C
57 C          END OF CLAUSE
58 C          -------------
59         LEVEL = LEVEL - 1
60         IF (LEVEL.LT.0) GO TO 70
61         RETURN
62 C
63 C          END OF FILE
64 C          -----------
65 50      M = M + 1
66         IF (M.GT.MAXLEX) GO TO 60
67         Q(M) = 54
68         DSPGLB(NOB+1) = DSPGLB(NOB)
69         RETURN
70 C
71 C          Q IS FULL
72 C
73 60      ERRCOD = 3
74         CALL ERR(ERRCOD)
75         RETURN
76 C
77 C          NO END CLAUSES ENOUGH
78 C
79 70      ERRCOD = 9
80         CALL ERR(ERRCOD)
81         RETURN
82         END
```

```
      SUBROUTINE GAMMA
C     ----------------
C        INTRODUCE GAMMA FOR GLOBAL VARIABLES
C
      COMMON/LIMITS/MAXLEX,MAX1,MAX2,MAX3,MAX4,MAX5,MAX6,MAX7,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/BUFFER/Q(216),P(216),UNUSED(68)
      COMMON/INDEX/L,M
      COMMON/GLB/GLBNST(2,80),GLBTAB(80),IGLB,DSPGLB(20)
      COMMON/CLPOS/LEVEL,NOB
      COMMON/LSTNOB/LSTNOB(5)
      INTEGER GLBNST,DSPGLB,GLBTAB,ERRCOD,SEARCH,P,Q,UNUSED

      I1 = LSTNOB(LEVEL)
      I2 = I1 + 1
      I2 = DSPGLB(I2)
      IF (I2.EQ.0) I2 = IGLB
      I = SEARCH(P(L),DSPGLB(I1),I2,GLBTAB)
      IF (I.EQ.0) GO TO 20

      IF (GLBNST(2,I).EQ.0) GO TO 10

C
C        INTRODUCE LATEST
C        ----------------

      M = M + 2
      IF (M.GT.MAXLEX) GO TO 30
      Q(M) = 4
      Q(M-1) = GLBNST(2,I) + 1000

C
C        INTRODUCE GAMMA
C        ---------------

10    M = M + 3
      IF (M.GT.MAXLEX) GO TO 30
      Q(M) = P(L)
      Q(M-1) = 3
      Q(M-2) = GLBNST(1,I) + 1000
      RETURN

C
C        LOCAL VARIABLE
C        --------------

20    M = M + 1
      IF (M.GT.MAXLEX) GO TO 30
      Q(M) = P(L)
      RETURN
C
C        Q IS FULL
C
30    ERRCOD = 3
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE CLAUSE
C     -----------------
C        ANALYSE A CLAUSE HEAD
C
      COMMON/LIMITS/MAXLEX,MAX1,MAXB,MAXEND,MAX4,MAX5,MAX6,MAX7,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/BUFFER/Q(216),P(216),UNUSED(68)
      COMMON/INDEX/L,M
      COMMON/CLPOS/LEVEL,NOB
      COMMON/NOARG/NOARG
      COMMON/CLTAB/CLTAB(2,20),DSPCL(2,20)
      COMMON/CLCAT/CLCAT(3,20)
      COMMON/GLB/GLBNST(2,80),GLBTAB(80),IGLB,DSPGLB(20)
      COMMON/LSTNOB/LSTNOB(5)
      INTEGER CLTAB,DSPCL,CLCAT,GLBNST,GLBTAB,DSPGLB,ERRCOD,P,Q,UNUSED
      INTEGER CLCALL

      NOARG = 0
      ERRCOD = 10
      M = M + 2
      IF (M.GT.MAXLEX) GO TO 30
      Q(M) = P(2)
      Q(M-1) = P(1)
      IF (.NOT.(
     1    (((P(1).EQ.16.OR.P(1).EQ.22).AND.(P(3).EQ.56.AND.P(4).EQ.42))
     2    .OR.(P(1).EQ.12.OR.P(1).EQ.25))
     3    .AND.(P(2).GT.63.AND.P(2).LT.1000))) CALL ERR(ERRCOD)
      NOB = NOB + 1
      IF (NOB.GT.MAXB) GO TO 50
      DSPGLB(NOB) = IGLB
      DSPGLB(NOB+1) = 0
      IF (P(L+2).EQ.56) CALL ARG
      IF (P(L+2).EQ.24) CALL GLOBAL
      M = M + 1
      IF (M.GT.MAXLEX) GO TO 30
      Q(M) = 53
C        BUILD UP THE TREE REPRESENTING THE CLAUSES
C        ------------------------------------------
      ERRCOD = 35
      IF (CLCALL(P(2)+500,LEVEL).GT.0) CALL ERR(ERRCOD)
      CLTAB(1,NOB) = P(2)
      IF (IGLB.GT.DSPGLB(NOB)) NOARG = NOARG + 100
      CLTAB(2,NOB) = NOARG
      CLCAT(1,NOB) = P(1)
      I1 = LSTNOB(LEVEL)
      LEVEL = LEVEL + 1
      IF (LEVEL.GT.MAXEND) GO TO 40
      IF (DSPCL(1,I1).EQ.0) GO TO 10
      I2 =  LSTNOB(LEVEL)
      DSPCL(2,I2) = NOB
```

```
      GO TO 20
10    DSPCL(1,I1) = NOB
20    LSTNOB(LEVEL) = NOB
      DSPCL(1,NOB) = 0
      DSPCL(2,NOB) = 0
      RETURN
C
C        Q IS FULL
C
30    ERRCOD = 3
      CALL ERR(ERRCOD)
      RETURN
C
C        TOO MUCH NESTING
C
40    ERRCOD = 12
      CALL ERR(ERRCOD)
      RETURN
C
C        TOO MANY BLOCKS
C
50    ERRCOD = 11
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE ARG
      --------------
C
C        ANALYSE A FORMAL PARAMETER LIST
C
      COMMON/LIMITS/MAXLEX,MAX1,MAX2,MAX3,MAX4,MAXARG,MAX6,MAX7,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/BUFFER/Q(216),P(216),UNUSED(68)
      COMMON/INDEX/L,M
      COMMON/NOARG/NOARG
      COMMON/ARG/ARGTAB(10),IARG
      INTEGER ARGTAB,ERRCOD,SEARCH,P,Q,UNUSED

      L = L + 4
      IARG = 1

      ERRCOD = 41
      IF (.NOT.(P(L).GT.63.AND.P(L).LT.1000)) CALL ERR(ERRCOD)
10    ARGTAB(IARG) = P(L)
      M = M + 1
      IF (M.GT.MAXLEX) GO TO 30
      Q(M) = P(L)
      ERRCOD = 19
      I1 = 1
      IF (SEARCH(P(L),I1,IARG,ARGTAB).NE.0) CALL ERR(ERRCOD)
      IARG = IARG + 1
      IF (IARG.GT.MAXARG) GO TO 40
      IF (.NOT.((P(L).GT.63.AND.P(L).LT.1000).AND.P(L+1).EQ.44))
     1                        GO TO 20
      L = L + 2
      GO TO 10

20    NOARG = IARG - 1
      IF ((P(L).GT.63.AND.P(L).LT.1000).AND.P(L+1).EQ.43) RETURN
      ERRCOD = 17
      CALL ERR(ERRCOD)
      RETURN
C
C        Q IS FULL
C
30    ERRCOD = 3
      CALL ERR(ERRCOD)
      RETURN
C
C        ARGTAB IS FULL
C
40    ERRCOD = 15
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE GLOBAL
C     --------------
C        ANALYSE A GLOBAL LIST
C
      COMMON/LIMITS/MAX0,MAX1,MAX2,MAX3,MAXGLB,MAX5,MAX6,MAX7,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/BUFFER/Q(216),P(216),UNUSED(68)
      COMMON/INDEX/L,M
      COMMON/CLPOS/LEVEL,NOB
      COMMON/GLB/GLBNST(2,80),GLBTAB(80),IGLB,DSPGLB(20)
      COMMON/ARG/ARGTAB(10),IARG
      INTEGER GLBNST,GLBTAB,DSPGLB,SEARCH,ERRCOD,P,Q,UNUSED

      L = L + 3

      ERRCOD = 42
      IF (.NOT.(P(L).GT.63.AND.P(L).LT.1000)) CALL ERR(ERRCOD)
10    GLBTAB(IGLB) = P(L)
      CALL UPDATE(P(L),P(1))
      ERRCOD = 18
      IF (SEARCH(P(L),DSPGLB(NOB),IGLB,GLBTAB).NE.0) CALL ERR(ERRCOD)
      ERRCOD = 17
      I1 = 1
      IF (SEARCH(P(L),I1,IARG,ARGTAB).NE.0) CALL ERR(ERRCOD)
      IGLB = IGLB + 1
      IF (IGLB.GT.MAXGLB) GO TO 30
      IF (.NOT.((P(L).GT.63.AND.P(L).LT.1000).AND.P(L+1).EQ.44))
     1                        GO TO 20
      L = L + 2
      GO TO 10

20    IF ((P(L).GT.63.AND.P(L).LT.1000).AND.P(L+1).EQ.53) RETURN
      ERRCOD = 14
      CALL ERR(ERRCOD)
      RETURN
C
C        THE GLOBAL STACK IS FULL
C
30    ERRCOD = 9
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE UPDATE(NAME,CLTYP)
C     -----------------------------
C        UPDATE THE TABLE REPRESENTING THE NESTING OF GLOBALS
C
      COMMON/GLB/GLBNST(2,80),GLBTAB(80),IGLB,DSPGLB(20)
      COMMON/CLPOS/LEVEL,NOB
      COMMON/LSTNOB/LSTNOB(5)
      INTEGER GLBTAB,DSPGLB,CLTYP,GLBNST,SEARCH

      IF (LEVEL.EQ.1) GO TO 10

      I1 = LSTNOB(LEVEL-1)
      I2 = LSTNOB(LEVEL)
      I = SEARCH(NAME,DSPGLB(I1),DSPGLB(I2),GLBTAB)
      IF (I.EQ.0) GO TO 10

      GLBNST(1,IGLB) = GLBNST(1,I) + 1
      GLBNST(2,IGLB) = GLBNST(2,I)
      IF (CLTYP.EQ.12.OR.CLTYP.EQ.22)
     1             GLBNST(2,IGLB) = GLBNST(2,I) + 1
      RETURN

10    GLBNST(1,IGLB) = 1
      GLBNST(2,IGLB) = 0
      IF (CLTYP.EQ.12.OR.CLTYP.EQ.22)
     1             GLBNST(2,IGLB) = 1
      RETURN
      END
```

```
      SUBROUTINE LEXDRI
C     -----------------
C        LEXDRI PRODUCE THE LEXICALLY ANALYSED VERSION OF P (LUSCII
C        FORM) INTO Q
C
      COMMON/LIMITS/MAXLEX,MAX1,MAX2,MAX3,MAX4,MAX5,MAX6,MAX7,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/BUFFER/P(216),Q(216),UNUSED(68)
      COMMON/INDEX/L,M
      INTEGER P,Q,UNUSED,ERRCOD

      L = 1
      M = 1

      CALL PROLUS

1     N = P(L) + 1
      GO TO (10,10,10,10,10,10,10,10,10,10,
     1       36,36,36,36,36,36,36,36,36,36,
     2       36,36,36,36,36,36,36,36,36,36,
     3       36,36,36,36,36,36,36,66,64,39,
     4       64,64,42,64,64,64,64,64,64,64,
     5       64,64,64,53,54,64,64,64,64,64,
     6       64,64,64,64,64                ),N

10    CALL NUMBER
      GO TO 1

36    CALL ALPHNM
      GO TO 1

39    CALL MINUS
      GO TO 64

42    CALL LFRDBR
      GO TO 1

53    Q(M) = 53
      RETURN

54    Q(M) = 54
      RETURN

64    Q(M) = P(L)
      M = M + 1
      IF (M.GT.MAXLEX) GO TO 70
66    L = L + 1
      GO TO 1

C
C        Q IS FULL
C
70    ERRCOD = 3
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE NUMBER
C     -----------------
C        HANDLE NUMBERS
C
      COMMON/LIMITS/MAXLEX,MAX1,MAX2,MAX3,MAX4,MAX5,MAX6,MAX7,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/BUFFER/P(216),Q(216),UNUSED(68)
      COMMON/INDEX/L,M
      COMMON/PARAM/PAR1,PAR2,PAR3,NLGHT,PAR5,PAR6
      INTEGER PAR1,PAR2,PAR3,PAR5,ERRCOD,P,Q,UNUSED,PAR6

      N = 0

10    N = 10 * N + P(L)
      IF (N.GT.NLGHT) GO TO 20
      L = L + 1
      IF (P(L).LE.9) GO TO 10

      Q(M) = N + 1000
      M = M + 1
      IF (M.GT.MAXLEX) GO TO 30
      RETURN
C
C        NUMBER TOO BIG
C
20    ERRCOD = 5
      CALL ERR(ERRCOD)
      RETURN
C
C        Q IS FULL
C
30    ERRCOD = 3
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE ALPHNM
C     -----------------
C        HANDLE ALPHANUMERIC NAMES
C
      COMMON/LIMITS/MAXLEX,MAX1,MAX2,MAX3,MAX4,MAX5,MAX6,MAX7,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/BUFFER/P(216),Q(216),UNUSED(84)
      COMMON/INDEX/L,M
      COMMON/PARAM/PAR1,PAR2,PAR3,PAR4,MAXSTG,PAR6
      INTEGER PAR1,PAR2,PAR3,PAR4,ERRCOD,CODE,P,Q,UNUSED,PAR6

      CODE = 0

10    CODE = 100 * CODE + P(L)
      L = L + 1
      IF (CODE.GT.MAXSTG) GO TO 40
      IF (P(L).LE.36) GO TO 10

20    CALL RESWRD(CODE,IDN)
      IF (IDN.EQ.28.OR.IDN.EQ.29) GO TO 30
      IF (IDN.EQ.0) CALL NAMTAB(CODE,IDN)
      Q(M) = IDN
      M = M + 1
      IF (M.GT.MAXLEX) GO TO 70
      RETURN

C
C        RESERVED WORD IS ''TRUE'' OR ''FALSE''
C        --------------------------------------
C
30    M = M + 1
      IF (M.GT.MAXLEX) GO TO 70
      Q(M) = ABS(IDN-28) + 1000
      RETURN
C
C        STRING TRUNCATED
C
40    ERRCOD = 6
      CALL ERR(ERRCOD)
50    IF (P(L).GT.36) GO TO 20
      L = L + 1
      GO TO 50

C
C        Q IS FULL
C
70    ERRCOD = 3
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE RESWRD(CODE,J)
C     -------------------------
C        TEST IF CODE (PACKED NAME) IS A RESERVERD WORD
C
      COMMON/RES/RESTAB(32),OPCODE(32),MAXRES
      INTEGER CODE,RESTAB,OPCODE

      DO 10 K = 1,MAXRES
      IF (RESTAB(K).EQ.CODE) GO TO 20
10    CONTINUE

      J = 0
      RETURN

20    J = OPCODE(K)
      RETURN
      END
```

```
      SUBROUTINE NAMTAB(CODE,J)
C     ------------------------
C        PUT CODE (PACKED NAME) IN AN IDENTIFICATION TABLE IF NOT IN IT
C
      COMMON/LIMITS/MAX0,MAXIDN,MAX2,MAX3,MAX4,MAX5,MAX6,MAX7,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/IDN/IIDN,IDNTAB(100)
      INTEGER CODE,ERRCOD

      DO 10 K = 1,IIDN
      IF (IDNTAB(K).EQ.CODE) GO TO 20
10    CONTINUE

      IIDN = IIDN + 1
      IF (IIDN.GT.MAXIDN) GO TO 30
      IDNTAB(IIDN) = CODE
      K = IIDN

20    J = K + 63
      RETURN
C
C        IDNTAB IS FULL
C
30    ERRCOD = 7
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE MINUS
C     ----------------
C        MAKE FROM A ONE ARITY MINUS A BINARY ONE
C
      COMMON/LIMITS/MAXLEX,MAX1,MAX2,MAX3,MAX4,MAX5,MAX6,MAX7,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/BUFFER/P(216),Q(216),UNUSED(68)
      COMMON/INDEX/L,M
      INTEGER P,Q,UNUSED,ERRCOD

      I1 = M - 1
      IF(.NOT.(Q(I1).LE.63.AND.Q(I1).NE.43)) RETURN

      Q(M) = 1000
      M = M + 1
      ERRCOD = 3
      IF (M.GT.MAXLEX) CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE LFRDBR
C     -----------------
C        INSERT A "56" (CLAUSE APPLICATION) FOR DEFINITIONS OR CALLS
C        OF CLAUSES WITH ARGUMENTS
C
      COMMON/LIMITS/MAXLEX,MAX1,MAX2,MAX3,MAX4,MAX5,MAX6,MAX7,MAX8,
     1              MAX9,MAX10,MAX11
      COMMON/BUFFER/P(216),Q(216),UNUSED(68)
      COMMON/INDEX/L,M
      INTEGER P,Q,UNUSED,ERRCOD

      IF (M.EQ.1) GO TO 10

      I1 = M - 1
      IF (Q(I1).NE.43.AND.Q(I1).LE.63) GO TO 10

      Q(M) = 56
      M = M + 1
      IF (M.GT.MAXLEX) GO TO 30

10    Q(M) = 42
      M = M + 1
      IF (M.GT.MAXLEX) GO TO 30
      L = L + 1
      RETURN
C
C        Q IS FULL
C
30    ERRCOD = 3
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
      SUBROUTINE PROLUS
C     -----------------
C        TRANSLATE IN LUSCII WHAT WAS READ IN EBCDIC
C
      COMMON/LIMITS/MAXLEX,MAX1,MAX2,MAX3,MAX4,MAX5,MAX6,MAX7,MAX8,
     1             MAX9,MAX10,MAX11
      COMMON/BUFFER/Q(216),UNUSED(204),P(80)
      COMMON/LINEN/LINEN
      INTEGER P,Q,UNUSED,CONVEL,ERRCOD

      M = 1
C
C        TRANSLATION
C        -----------
10    I2 = M + 72
      IF (I2.GT.MAXLEX) GO TO 60
      CALL INOUT
      LINEN = LINEN + 1
      DO 20 K = M,I2
      I1 = 1 + K - M
      I1 = P(I1)
20    Q(K) = CONVEL(I1)
C
C        END OF FILE
C        -----------
30    I2 = I2 - 1
      IF (Q(I2).NE.37) GO TO 40

      IF (I2.GT.M) GO TO 30

      Q(M) = 54
      RETURN
C
C        END OF SENTENCE
C        ---------------
40    M = M + 72
      IF (Q(M).EQ.12) GO TO 50
      Q(I2+1) = 53
      RETURN
C
C        CONTINUATION CARD
C        -----------------
50    M = I2 + 1
      GO TO 10
```

```
C
C         Q IS FULL
C
60      ERRCOD = 3
        CALL ERR(ERRCOD)
        RETURN
        END
```

```
 1        INTEGER FUNCTION CONVEL(CARACT)
 2 C      --------------------------------
 3 C         CONVERT "CARACT" FROM EBCDIC TO LUSCII
 4 C
 5        COMMON/EBC/EBCDIC(64)
 6        INTEGER EBCDIC,CARACT,ERRCOD
 7
 8        DO 10 K = 1,64
 9        IF (EBCDIC(K).EQ.CARACT) GO TO 20
10 10     CONTINUE
11
12        K = 52
13        ERRCOD = 4
14        CALL ERR(ERRCOD)
15
16 20     CONVEL = K - 1
17        RETURN
18        END
```

```
      SUBROUTINE INOUT
C     ----------------
C        READ AND PRINT OUT THE LUCID SOURCE PROGRAM
C
      COMMON/BUFFER/UNUSED(420),Q(80)
      COMMON/PARAM/FLINS,FLOUTS,PAR3,PAR4,PAR5,PAR6
      INTEGER PAR3,PAR4,PAR5,FLOUTS,FLINS,Q,UNUSED,PAR6

      READ(FLINS,1) (Q(K),K=1,80)
1     FORMAT(80A1)

      WRITE(FLOUTS,2) (Q(K),K=1,80)
2     FORMAT(1X,80A1)

      RETURN
      END
```

```
      INTEGER FUNCTION SEARCH(NAME,LL,UL,TAB)
C     ---------------------------------------
C        STANDARD ROUTINE
C
      COMMON/STDFOR/I1,I2
      INTEGER TAB(1),UL

      IF (UL.LE.LL) GO TO 20
      I1 = UL - 1
      DO 10 K = LL,I1
      IF (TAB(K).EQ.NAME) GO TO 30
10    CONTINUE

20    SEARCH = 0
      RETURN

30    SEARCH = K
      RETURN
      END
```

```
      INTEGER FUNCTION CLCALL(NAME,LEVEL)
C     -----------------------------------
C        EQUIVALENT TO SEARCH BUT FOR AN INDEXED SEQUENTIAL TABLE
C
      COMMON/CLTAB/CLTAB(2,20),DSPCL(2,20)
      COMMON/LSTNOB/LSTNOB(5)
      INTEGER CLTAB,DSPCL

      I1 = LSTNOB(LEVEL)
      I = DSPCL(1,I1)

10    IF (I.EQ.0) GO TO 30

      IF (CLTAB(1,I).EQ.NAME) GO TO 20

      I = DSPCL(2,I)
      GO TO 10

20    CLCALL = I
      RETURN

30    CLCALL = 0
      RETURN
      END
```

```
      SUBROUTINE CONTRL
C     -----------------
C        CONTROL IF THE NB OF ACTUAL PARA = NB OF FORMAL PARA,FOR EACH
C        CLAUSE CALL
C
      COMMON/BUFFER/OP(125),OPDE1(125),OPDE2(125),LUCCAT(20),UNUSED(105)
      COMMON/CLCAT/CLCAT(3,20)
      COMMON/CLPOS/LEVEL,NOB
      INTEGER OP,OPDE1,OPDE2,UNUSED,CLCAT,CLNAM

      DO 20 K =1,NOB
      I= CLCAT(2,K)
      IF (I.EQ.0) GO TO 20

10    IF (OP(I).NE.2) GO TO 20
      NAME = OPDE1(I) - 500
      IF (NAME.LT.0) NAME = CLNAM(I,K)

      IF (NAME.NE.0) CALL TSTEQN(NAME,K,I)
      I = I + 1
      GO TO 10
20    CONTINUE
      RETURN
      END
```

```
      INTEGER FUNCTION CLNAM(I,K)
C     ---------------------------
C        SEARCH THE NAME OF THE CLAUSE CALLED
C
      COMMON/BUFFER/OP(125),OPDE1(125),OPDE2(125),LUCCAT(20),UNUSED(105)
      INTEGER OP,OPDE1,OPDE2,UNUSED,ERRCOD

      ILUC = LUCCAT(K) - 1
      K1 = OPDE1(I) + ILUC
      IF (OP(K1).EQ.3) GO TO 20
      IF (OP(K1).EQ.1) GO TO 10

C        ERROR SOMEWHERE?
C        ----------------
C
      ERRCOD = 36
      CALL ERR(ERRCOD)
      CLNAM = 0
      RETURN

C        NAME AS PARAMETER
C        -----------------
C
10    ERRCOD = 37
      CALL ERR(ERRCOD)
      CLNAM = 0
      RETURN

C        NAME AS GLOBAL
C        --------------
C
20    IF (OPDE2(K1).GE.500) GO TO 30
      ERRCOD = 38
      CALL ERR(ERRCOD)
      CLNAM = 0
      RETURN

30    CLNAM = OPDE2(K1) - 500
      RETURN
      END
```

```
      SUBROUTINE TSTEQN(NAME,K,I)
C     ---------------------------
C        COUNTS THE NB OF ACTUAL PARA AND COMPARE IT TO THE NB OF
C        FORMAL PARAMETERS
C
      COMMON/BUFFER/OP(125),OPDE1(125),OPDE2(125),LUCCAT(20),UNUSED(105)
      COMMON/CLTAB/CLTAB(2,20),DSPCL(2,20)
      COMMON/NOARG/NOARG
      INTEGER OP,OPDE1,OPDE2,UNUSED,CLTAB,DSPCL,ERRCOD

      ILUC = LUCCAT(K) - 1
      NOARG = 0
      K = I
10    II = OPDE2(K)
      IF (II.EQ.700) GO TO 20
      NOARG = NOARG + 1
      K = II + ILUC
      GO TO 10

20    N1 = CLTAB(2,NAME)
      IF (N1.GE.100) N1 = N1 - 100
      IF (NOARG.EQ.N1) RETURN
      ERRCOD = 39
      CALL ERR(ERRCOD)
      RETURN
      END
```

```
    1       SUBROUTINE BADDEF(LEVEL)
    2 C     ------------------------
    3 C     SEARCH FOR INCONSISTENT VARIABLE DEFINTIONS
    4 C
    5       COMMON/REN/RENTAB(2,100),IREN,DSPREN(20),NEWNAM(100)
    6       INTEGER ERRCOD,RENTAB,DSPREN
    7
    8       I1 = DSPREN(LEVEL)
    9       DO 40 K = I1,IREN
   10       IF (RENTAB(1,K)-1)10,20,30
   11
   12 C        VARIABLE NOT DEFINED
   13 C        --------------------
   14 C
   15 10    ERRCOD = 32
   16       CALL ERR(ERRCOD)
   17       CALL DBLDEF(NEWNAM(K))
   18       GO TO 40
   19
   20 C        VARIABLE DEFINED BUT NOT USED
   21 C        -----------------------------
   22 C
   23 20    ERRCOD = 33
   24       IF (RENTAB(2,K).GT.0) GO TO 40
   25       CALL ERR(ERRCOD)
   26       CALL DBLDEF(NEWNAM(K))
   27       GO TO 40
   28
   29 C        MORE THEN ONE DEFINITION
   30 C        ------------------------
   31 30    CALL DBLDEF(NEWNAM(K))
   32       ERRCOD = 34
   33       CALL ERR(ERRCOD)
   34
   35 40    CONTINUE
   36       RETURN
   37       END
```

```
      SUBROUTINE DBLDEF(NAME)
C     -----------------------
C        SEARCH FOR THE EARLIER EBCDIC CODE OF A DOUBLE DEFINED VARIABLE
C
      COMMON/IDN/IIDN,IDNTAB(100)
      COMMON/EBC/EBCDIC(64)
      COMMON/PARAM/PAR1,PAR2,PAR3,PAR4,PAR5,FLOUTR
      INTEGER NAM(4),EBCDIC,CODE,PAR1,PAR3,PAR4,PAR5,FLOUTR,PAR2

      I = 4
      CODE = IDNTAB(NAME-63)

10    R = CODE/100
      I1 = IFIX(R)
      I2 = CODE - 100 * I1
      NAM(I) = EBCDIC(I2)
      CODE = I1
      I = I - 1
      IF (I1.GE.100) GO TO 10

      NAM(I) = EBCDIC(I1)
      WRITE(FLOUTR,1) (NAM(J),J=I,4)
1     FORMAT(1X,4A1)
      RETURN
      END
```

```
      SUBROUTINE ERR(ERRCOD)
C     ----------------------

      COMMON/NERR/NERR
      COMMON/LINEN/LINEN
      COMMON/PARAM/PAR1,PAR2,PAR3,PAR4,PAR5,FLOUTR
      INTEGER ERRCOD,PAR1,PAR2,PAR3,PAR4,PAR5,FLOUTR

      GO TO (     10, 20, 30, 40, 50, 60, 70, 80, 90,
     1       100,110,120,130,140,150,160,170,180,190,
     2       200,210,220,230,240,250,260,270,280,290,
     3       300,310,320,330,340,350,360,370,380,390,
     4       400,410,420                                ), ERRCOD

10    WRITE(FLOUTR,1000)
1000  FORMAT(1X,'WELCOME TO THE LUCID TEAM.SEE YOU AT THE END OF',/,
     1          'THE TRANSFORMER ANALYSIS.HAVE A GOOD TIME AND',/,
     2          'GOOD LUCK.',/)
      RETURN

20    IF (NERR.GT.0) GO TO 25
      WRITE(FLOUTR,1100)
1100  FORMAT(1X,'CONGRATULATIONS ! YOU SEE IT WAS NOT SO DIFFICULT',/,
     1          'WE ARE GLAD TO HAVE YOU WITH US AND WE HOPE YOU WILL',/,
     2          'ENJOY IT. SEE YOU FOR THE RESULTS.',/)
      RETURN
25    WRITE(FLOUTR,1200)
1200  FORMAT(1X,'SORRY I CAN NOT TAKE THE RESPONSIBILITY TO',/,
     1          'LET YOU GO ON,BUT DO NOT LOOSE YOUR SELF CONTROL',/,
     2          'IT WILL BE ALRIGHT NEXT TIME. SEE YOU LATER',/,
     3          '              - THE LUCID TEAM-                ',/)
      RETURN

30    NERR = NERR + 1
      WRITE(FLOUTR,1300)
1300  FORMAT(1X,'INV INDEX: MAXLEX',/)
      GO TO 9999

40    NERR = NERR + 1
      WRITE(FLOUTR,1400) (LINEN)
1400  FORMAT(1X,'INV CHAR IN LINE NUMBER:',I5,/)
      RETURN

50    NERR = NERR + 1
      WRITE(FLOUTR,1500) (LINEN)
1500  FORMAT(1X,'NUMBER EXCEEDS 2**31',/)
      RETURN

60    WRITE(FLOUTR,1600) (LINEN)
1600  FORMAT(1X,'WARNING: STRING OF CHARACT EXCEEDS 4',/)
```

```
      RETURN

70    NERR = NERR + 1
      WRITE(FLOUTR,1700)
1700  FORMAT(1X,'INV INDEX: MAXIDN',/)
      GO TO 9999

80    NERR = NERR + 1
      WRITE(FLOUTR,1900) (LINEN)
1300  FORMAT(1X,'GLOBAL AS LEFT HAND SIDE IN LINE NUMBER:',I5,/)
      RETURN

90    NERR = NERR + 1
      WRITE(FLOUTR,1900) (LINEN)
1900  FORMAT(1X,'TOO MANY END CLAUSES.WE ARE IN LINE:',I5,/)
      RETURN

100   NERR = NERR + 1
      WRITE(FLOUTR,2000) (LINEN)
2000  FORMAT(1X,'INV CLAUSE HEAD IN LINE NUMBER:',I5,/)
      RETURN

110   NERR = NERR + 1
      WRITE(FLOUTR,2100)
2100  FORMAT(1X,'INV INDEX:MAXB',/)
      GO TO 9999

120   NERR = NERR + 1
      WRITE(FLOUTR,2200)
2200  FORMAT(1X,'INV INDEX:MAXEND',/)
      GO TO 9999

130   NERR = NERR + 1
      WRITE(FLOUTR,2300)
2300  FORMAT(1X,'INV INDEX:MAXGLB',/)
      GO TO 9999

140   NERR = NERR + 1
      WRITE(FLOUTR,2400) (LINEN)
2400  FORMAT(1X,'INV USING LIST IN LINE NUMBER:',I5,/)
      RETURN

150   NERR = NERR + 1
      WRITE(FLOUTR,2500)
2500  FORMAT(1X,'INV INDEX: MAXARG',/)
      GO TO 9999

160   NERR = NERR + 1
      WRITE(FLOUTR,2600) (LINEN)
2600  FORMAT(1X,'INV ARGUMENT LIST IN LINE NB :',I5,/)
```

```
101         RETURN
102
103 170     NERR = NERR + 1
104         WRITE(FLOUTR,2700) (LINEN)
105 2700    FORMAT(1X,'ARG AND GLB MIXED UP IN LINE NB :',I5,/)
106         RETURN
107
108 180     NERR = NERR + 1
109         WRITE(FLOUTR,2800) (LINEN)
110 2800    FORMAT(1X,'DOUBLE USE OF GLOBALS IN LINE NB :',I5,/)
111         RETURN
112
113 190     NERR = NERR + 1
114         WRITE(FLOUTR,2900) (LINEN)
115 2900    FORMAT(1X,'DOUBLE USE OF ARG IN LINE NB :',I5,/)
116         RETURN
117
118 200     NERR = NERR + 1
119         WRITE(FLOUTR,3000)
120 3000    FORMAT(1X,'END CLAUSES ARE MISSING',/)
121         RETURN
122
123 210     NERR = NERR + 1
124         WRITE(FLOUTR,3100) (LINEN)
125 3100    FORMAT(1X,'INV BEGIN OF SENTENCE IN LINE',I5,/)
126         RETURN
127
128 220     NERR = NERR + 1
129         WRITE(FLOUTR,3200)
130 3200    FORMAT(1X,'INV INDEX : MAXSYN',/)
131         GO TO 9999
132
133 230     NERR = NERR + 1
134         WRITE(FLOUTR,3300)
135 3300    FORMAT(1X,'INV INDEX : KVR',/)
136         GO TO 9999
137
138 240     NERR = NERR + 1
139         WRITE(FLOUTR,3400)
140 3400    FORMAT(1X,'INV INDEX : KOP',/)
141         GO TO 9999
142
143 250     NERR = NERR + 1
144         WRITE(FLOUTR,3500) (LINEN)
145 3500    FORMAT(1X,'NO OPERANDS ENOUGH IN LINE NB :',I5,/)
146         RETURN
147
148 260     NERR = NERR + 1
149         WRITE(FLOUTR,3600) (LINEN)
150 3600    FORMAT(1X,'INV OPERATOR IN LINE NB :',I5,/)
```

```
151         RETURN
152
153  270    NERR = NERR + 1
154         WRITE(FLOUTR,3700)
155  3700   FORMAT(1X,'INV INDEX : MAXCAL',/)
156         GO TO 9999
157
158  280    WRITE(FLOUTR,3800)
159  3800   FORMAT(1X,'WARNING : FUN OR MAP AS PARAMETER',/)
160         RETURN
161
162  290    NERR = NERR + 1
163         WRITE(FLOUTR,3900)
164  3900   FORMAT(1X,'CLAUSE NAME AS LEFT HAND SIDE',/)
165         RETURN
166
167  300    NERR = NERR + 1
168         WRITE(FLOUTR,4000)
169  4000   FORMAT(1X,'INV INDEX : MAXLUC',/)
170         GO TO 9999
171
172  310    NERR = NERR + 1
173         WRITE(FLOUTR,4100)
174  4100   FORMAT(1X,'INV INDEX : MAXPIC',/)
175         GO TO 9999
176
177  320    NERR = NERR + 1
178         WRITE(FLOUTR,4200)
179  4200   FORMAT(1X,'VAR NOT DEFINED',/)
180         RETURN
181
182  330    WRITE(FLOUTR,4300)
183  4300   FORMAT(1X,'VAR DEFINED BUT NOT USED',/)
184         RETURN
185
186  340    NERR = NERR + 1
187         WRITE(FLOUTR,4400)
188  4400   FORMAT(1X,'VAR DEFINED MORE THEN ONCE',/)
189         RETURN
190
191  350    NERR = NERR + 1
192         WRITE(FLOUTR,4500)
193  4500   FORMAT(1X,'CLAUSE DEFINED MORE THEN ONCE',/)
194         RETURN
195
196  360    NERR = NERR + 1
197         WRITE(FLOUTR,3600)
198  4600   FORMAT(1X,'ERR IN LUCODE PRODUCTION OR VERY BAD PROG. STRUCT',/)
199         RETURN
200
```

```
  201 370   WRITE(FLOUTR,4700)
  202 4700  FORMAT(1X,'NB OF PARA NOT CONTROLED. CLAUSE NAME AS PARA',/)
  203       RETURN
  204
  205 380   WRITE(FLOUTR,4800)
  206 4800  FORMAT(1X,'NB OF PARA NOT CONTROLLED. TOO COMPLICATED STRUCTURE',/
  207      1           'LG: GLOBAL NAME WAS PARA',/)
  208       RETURN
  209
  210 390   NERR = NERR + 1
  211       WRITE(FLOUTR,4900)
  212 4900  FORMAT(1X,'NB OF ACTUAL PAR IS DIFF FROM THE NB OF FORMAL PAR',/)
  213       RETURN
  214
  215 400   NERR = NERR + 1
  216       WRITE(FLOUTR,5000)
  217 5000  FORMAT(1X,'FUN OR PRO AS GLOBAL IN CMP OR MAP',/)
  218       RETURN
  219
  220 410   NERR = NERR + 1
  221       WRITE(FLOUTR,5100) (LINEN)
  222 5100  FORMAT(1X,'EMPTY ARG LIST IN LINE NB',I3,/)
  223       RETURN
  224
  225 420   NERR = NERR + 1
  226       WRITE(FLOUTR,5200) (LINEN)
  227 5200  FORMAT(1X,'EMPTY GLB LIST IN LINE NB',I3,/)
  228       RETURN
  229
  230 9999  WRITE(FLOUTR,9998)
  231 9998  FORMAT(1X,'ASK FOR HELP TO THE MANAGER',/)
  232       RETURN
  233       END
```

```
      BLOCK DATA
      COMMON/SYMB/SYMB
C        SYMBOL TABLE
C        ------------
C  A
C
C     ARGTAB(I)      ARGUMENT TABLE. CONTAINS THE ARG OF A CLAUSE (MAX=10)
C     ARIT           ARITY OF AN OPERATOR
C     ARITY(64)      TABLE CONTAINING THE ARITY OF ALL THE OPERATORS
C
C
C  C
C
C     CARACT         CHARACTER
C     CLCAT(I,J)     CLAUSE CATALOGUE
C                    I=1  CLAUSE TYPE
C                    I=2  ADDRESS OF THE FIRST CALL
C                    I=3  NUMBER OF CALLS
C                    J = NUMBER OF CLAUSE  (MAX=20)
C     CLTAB(I,J)     CLAUSE TABLE
C                    I=1  CLAUSE NAME
C                    I=2  NUMBER OF ARGUMENTS
C                         (NB OF ARG = REAL NB OF ARG + 100 IF THAT CLAUSE
C                          HAS GLOBALS)
C     CLTYP(I)       CLAUSE TYPE
C                    TYPE OF THE CURRENT CLAUSE. (STACK)
C                    I = LEVEL  (MAX=5)
C     CODE           CODE
C
C
C  D
C
C     DSPADD(I,J)    DISPLAY FOR A TABLE CONTAINING ADDRESSES
C                    I=1  BEGIN OF THE LIST OF CALL ADDRESSES IN STADD
C                    I=2  CURRENT END OF THAT LIST
C                    J  NUMBER OF CLAUSES (MAX=20)
C     DSPCL(I,J)     DISPLAY FOR CLTAB
C                    I=1  BEGIN OF THE LIST DEFINED BELOW
C                    I=2  LIST OF NESTING CLAUSES(AT SAME LEVEL) FOR A
C                         CLAUSE OF LEVEL J
C                    THIS REPRESENT THE LUCID PROGRAM AS A TREE WHERE
C                    THE CLAUSE DEFINITIONS ARE THE NODES
C     DSPGLB         DISPLAY FOR GLBTAB
C                    BEGIN OF GLOBAL LIST OF CLAUSE NUMBER I
C     DSPPIC(I,J)    DISPLAY OF A PICTURE OF THE PROGRAM
C                    I=1 BEGIN IN PICPRG OF PAIRS OF POINTERS FOR THE
C                    THE CLAUSE J
C                    I=2 CURRENT END OF THAT LIST OF PAIRS
C     DSPREN(I)      DISPLAY FOR THE RENAME TABLE
C                    BEGIN OF NEW IDENTIFICATION TABLE OF LEVEL I
```

```
C  E
C     EBCDIC(64)     GIVES THE LUSCII CODE FOR ALL EBCDIC CHARACTERS
C     ERRCOD         ERROR CODE
C  F
C     FLAG           LOGICAL VARIABLE
C     FLINS          FILE INPUT SOURCE
C     FLOUT          FILE OUTPUT LUCODE
C     FLOUTL
C     FLOUTS         FILE OUTPUT SOURCE
C     FLOUTR         FILE OUTPUT ERROR
C  G
C     GLBNST(I,J)    GLOBAL NESTING
C                    I=1  NB OF LEVELS TO GO BACK WITH GAMMA
C                    I=2  NB OF LEVELS TO GO BACK WITH LATEST
C                    J = POINTER IN GLBTAB  (MAX=80)
C     GLBTAB(J)      CONTAINS ALL THE GLOBAL LISTS (CLAUSE BY BLAUSE)
C  H
C     HGK            HIGHEST VALUE OF K (DURING THE REORDERING)
C  I
C     I              INDEX
C     I1,I2...II     TEMPORARY VARIABLES
C     IADD           POINTS AT THE TOP OF STADD
C     IARG           POINTS AT THE TOP OF ARGTAB
C     IDN            IDENTIFICATION
C     IDNTAB(I)      IDENTIFICATION TABLE  (MAX=100)
C     IGLB           POINTS AT THE TOP OF GLBTAB
C     IIDN           POINTS AT THE TOP OF IDNTAB
C     ILUC           BEGIN OF THE CLAUSE BODY OF "I"
C     IOP            POINTS AT THE TOP OF OPST
C     IPIC           POINTS AT THE TOP OF PICPRG
C     IREN           POINTS AT THE TOP OF RENTAB
C     IVR            POINTS AT THE TOP OF VRST
```

```
C  K
C
C     K              LOOP INDEX
C
C
C  L
C
C     L              CURRENT POINTER IN P
C     LENGHT         ZONE LENGHT TO COPY
C     LEVEL          LEVEL OF CLAUSE DEFINITION
C     LL             LOWER LIMIT
C     LINEN          LINE NUMBER
C     LSTM           LAST M
C     LSTNOB(I)      LAST NB OF BLOCK OF LEVEL I
C     LSTTYP(I)      LAST CLAUSE TYPE OF LEVEL I
C     LUCCAT(I)      LUCODE CATALOGUE
C                    GIVES THE BEGIN OF LUCODE FOR EACH CLAUSE
C                    I = NB OF CLAUSE  (MAX=20)
C
C
C  M
C
C     M              CURRENT POINTER IN Q
C     MAXARG         LIMIT NUMBER OF ARGUMENTS
C     MAXB           LIMIT NUMBER OF CLAUSES
C     MAXCAL         LIMIT NUMBER OF CALLS
C     MAXGLB         LIMIT NUMBER OF GLOBALS
C     MAXIDN         LIMIT NUMBER OF VARIABLES IN LUCODE
C     MAXLUC         LIMIT NUMBER OF LUCODE EQUATIONS
C     MAXOPC         LIMIT NUMBER OF OPERATORS IN A LUCID SENTENCE
C     MAXVR          LIMIT NUMBER OF VARIABLES IN A LUCID SENTENCE
C     MAXPIC         LIMIT OF PICPRG
C     MAXRES         LIMIT NUMBEROF RESERVED WORDS
C     MAXEND         LIMIT NUMBER OF NESTING
C     MAXSTG         MAX STRING LENGHT
C     MAXSYN         LIMIT FOR THE REPRESENTATION OF THE ALL PROGRAM
C     MAXLEX         MAX LENGHTF A LUSCII SENTENCE
C
C
C  N
C     N              NUMBER  (PACKED NAME IN TLE LEXICAL ANALYSER)
C     N1             TEMPORARY VARIABLE
C     NAME           NAME OF A VARIABLE - CLAUSE
C     NERR           NUMBER OF ERRORS
C     NEW            NEW NAME
C     NEWNAM(I)      "IDENTIFICATION" TABLE DURING THE RENAMING
C     NLENGHT        NUMBER LENGHT
C     NOB            NB OF BLOCK
C     NOARG          NB OF ARGUMENTS
C
```

```
C  O
C     OP(I)          OPERATOR OF THE ITH LUCODE EQUATION
C     OPDE1(I)       FIRST OPERAND OF THAT EQUATION
C     OPDE2          SECOND OPERAND OF THAT EQUATION
C     OPCODE         OPERATOR CODE
C     OPST(I)        OPERATOR STACK  (MAX=25)

C  P
C     PICPRG(I,J)    PICTURE OF THE PROGRAM
C                    I=1  BEGIN OF A PART A OF A CLAUSE DEFINITION
C                    I=2  END OF IT
C                    I=3  MAKE A LIST,FOR EACH CLAUSE,OF THE PAIRS OF
C                         POINTERS DEFINED ABOVE
C                    DEFINED ABOVE
C     PREC(64)       TABLE CONTAINING THE PRIORITY OF THE OPERATORS

C  R
C     RENTAB(I,J)    RENAME TABLE
C                    I=1  NB OF DEFINITION EQUATIONS FOR THE VAR GIVEN BY
C                    NEWNAM(J)
C                    I=2  VAR USED OR NOT
C                    J = NB OF CLAUSE

C  S
C     STADD(I,J)     I=1  ADDRESS OF A CALL IN THE PROGRAM
C                    I=2  POINTS IN STADD. GIVES THE NEXT CALL ADDRESS

C  T
C     TEMPVR         TEMPORARY VARIABLE
C     TAB            TABLE  (GIVEN AS PARAMETER)
C     TOKEN          ELEMENT TOKEN  (SYNT ANALYSER)

C  U - V
C     UNUSED         UNUSED MEMORY
C     UL             UPPER LIMIT

C     VRST           VARIABLE STACK
      END
```

```
      BLOCK DATA
      COMMON/MODIF/MODIF
C        NOTE: MODIFY FOR EACH ERROR IN BLOCK DATA (LIMITS)
C     MAXLEX:   MODIFY DIM OF P,Q,UNUSED IN
C                    LEXDRI,NUMBER,ALPHNM,MINUS,LFRDBR,PROLUS,INOUT,ARG,
C                    CLAUSE,GAMMA,OPDRI
C           NOTE:    SUM OF DIM OF P,Q,UNUSED MUST BE = TO MAXSYN
C                    SO MODIFY ALSO AS FOR MAXSYN
C
C     MAXIDN:   MODIFY DIM OF - IDNTAB IN NAMTAB,B.D. ,TYPE,EMIT
C                             - RENTAB-NEWNAM IN B.D,ENDCL,ADELE,ADNAM,
C                                                BADDEF,DBLDEF
C
C     MAXB:     MODIFY DIM OF
C                  CLTAB - DSPCL IN CLAUSE,CLCALL,B.D.,TSTEQN
C                  CLCAT IN FUNMAP,LOCCL,GLBCL,B.D.,RENCAL,CONTRL,TRSDRI
C                  DSPGLB IN NIL,OPDRI,GAMMA,CLAUSE,GLOBAL,UPDATE,B.D.,
C                            ADGLB
C                  DSPADD IN CLHEAD,UPDCAL,B.D,RENCAL
C                  DSPPIC SEE PICPRG IN MAXPIC
C
C     MAXEND:   MODIFY DIM OF
C                  LSTTYP IN BETA,CLHEAD,GLBCL
C                  LSTNOB IN BETA,CLHEAD,ENDCL,FUNMAP,UPDCAL,LOCCL,NIL,
C                            OPDRI,GAMMA,CLAUSE,UPDATE,CLCAL,B.D.,RENDRI,
C                            PHBGOB,PHEOB
C     MAXGLB:   MODIFY DIM OF GLBNST,GLBTAB IN (SEE DSPGLB IN MAXB)
C
C     MAXARG:   MODIFY DIM OF ARGTAB IN GLOBAL,ARG
C
C     MAXSYN:   MODIFY AS FOR MAXLEX(SEE SPECIAL NOTE IN MAXLEX) +
C                  P,Q IN BETA,CLHEAD,EQUATN,COPY,TSTLOC,TSTGLB,FUNMAP,
C                         LOCCL,GLBCL,SYNAN,TYPE,RGHDSD,EXP,CODGEN,EMIT,
C                         TRSDRI,RENDRI,RENCAL,RENVR,ADNAM,CONTRL,TSTEQN
C
C     MAXVR-KVR: MODIFY VRST IN EXP,CODGEN,EMIT,COMMA
C
C     MAXOP-KOP: MODIFY DIM OF OPST AS FOR VRST (SEE MAXVR)
C
C     MAXCAL:   MODIFY STADD AS FOR DSPPAD (SEE MAXB)
C
C     MAXLUC:   MODIFY DIM OF P,Q AS FOR MAXSYN-MAXLEX
C           NOTE: -B.D. = BLOCK DATA INITIAL. THOSE TABLES
C                  -IF MODIFY RESTAB (RESERVED WORDS) MOD. DIM. OF RESTAB
C                   IN RESWRD
      END
```

Lucid diagnostics

1
2
3 inv index MAXLEX
4 inv charact
5 nb exceeds 2**31
6 string of charact. excedds 4 (W)
7 inv index MAXIDN
8 global as left hand side
9 too many end clauses
10 inv clause head
11 inv index MAXB
12 inv index MAXEND
13 inv index MAXGLB
14 inv using list
15 inv index MAXARG
16 inv argument list
17 arg and glb mixed up
18 double use of globals
19 double use of arguments
20 end clauses missing
21 inv. begin of sentence
22 inv. index MAXSYN
23 inv. index KVR
24 inv. index KOP
25 no operands enough
26 inv. operator
27 inv. index MAXCAL
28 fun or map as para (W)
29 clause name as left hand side
30 inv. index MAXLUC
31 inv. index MAXPIC
32 var. not defined
33 var. def. but not used (W)
34 more then one def.
35 clause defined more then once
36 error in Lucode product or very bad progr. struct.
37 clause name is para - no control (W)
38 complicated struct. - no control (W) (e.g. global clause name is also para)
39 nb of actual para ≠ nb of formal para
40 fun or map as glb in map-cmp.

```
      BLOCK DATA
      COMMON/EBC/EBCDIC(64)
      INTEGER EBCDIC

      DATA EBCDIC /'0','1','2','3','4','5','6','7','8','9',
     1            'A','B','C','D','E','F','G','H','I','J',
     2            'K','L','M','N','O','P','Q','R','S','T',
     3            'U','V','W','X','Y','Z',' ',' ','+','-',
     4            '*','/','(',')',',','.','=','$','!',';',
     5            ':','?','|','¬','¬','=','@','¬','<','>',
     6            '#','&','%','$'/
      END
```

```
1         BLOCK DATA
2         COMMON/NERR/NERR
3         COMMON/LINEN/LINEN
4
5         DATA NERR,LINEN/0,0/
6         END
```

```
      BLOCK DATA
      COMMON/PARAM/FLINS,FLOUTS,FLOUTL,NLGHT,MAXSTG,FLOUTR
      INTEGER FLINS,FLOUTS,FLOUTL,FLOUTR

      DATA FLINS,FLOUTS,FLOUTL,NLGHT,MAXSTG,FLOUTR/5,25,26,2147483640,
     1            363636,27                                          /
      END
```

```
      BLOCK DATA
      COMMON/RES/RESTAB(32),OPCODE(32),MAXRES
      INTEGER RESTAB,OPCODE

      DATA MAXRES/32/
      DATA RESTAB/102810,122225,142313,152829,153023,1815,221025,
     1            233329,302816,252724,232429,15102128,29273014,
     2            30252423,32171423,
     3            2427,151134,1426,14212814,29171423,102313,25243227,
     4            143325,2123,

     5            12242225,15182728,15302312,22102525,23143329,
     6            30281823,25272413,25243214/

      DATA OPCODE/10,12,13,15,16,18,22,
     1            23,24,25,27,28,29,
     2            30,32,
     3            33,34,35,49,50,61,62,
     4            60,63,

     5            12,15,16,22,23,
     6            24,25,62/
      END
```

```
      BLOCK DATA
      COMMON/LIMITS/MAXLEX,MAXIDN,MAXB,MAXEND,MAXGLB,MAXARG,MAXSYN,
     1              MAXVR,MAXOP,MAXCAL,MAXLUC,MAXPIC

      DATA MAXLEX,MAXIDN,MAXB,MAXEND,MAXGLB,MAXARG,MAXSYN,MAXVR,MAXOP,
     1     MAXCAL,MAXLUC,MAXPIC/216,100,20,5,80,10,500,50,25,50,125,40/
      END
```

```
1         BLOCK DATA
2         COMMON/IDN/IIDN,IDNTAB(100)
3
4         DATA IIDN,IDNTAB(1),IDNTAB(2)/2,271428,243029/
5         END
```

```
1       BLOCK DATA
2       COMMON/CLPOS/LEVEL,NOB
3       COMMON/LSTNOB/LSTNOB(5)
4
5       DATA LEVEL,NOB/1,1/
6       DATA LSTNOB(1)/1/
7       END
```

```
      BLOCK DATA
      COMMON/ARITY/ARITY(64)
      INTEGER ARITY

      DATA ARITY/-1, 1,-1, 2, 2,-1,-1,-1,-1,-1,
     1            2,-1,-1,-1,-1, 1,-1,-1, 0,-1,
     2           -1,-1,-1, 1,-1,-1, 2, 1,-1,-1,
     3            2,-1, 2, 2, 2, 2,-1,-1, 2, 2,
     4            2, 2, 0, 0, 0,-1,-1,-1,-1, 2,
     5            2,-1,-1, 0,-1,-1, 2,-1, 2, 2,
     6            1, 2, 2, 1                    /
      END
```

```
      BLOCK DATA
      COMMON/PREC/PREC(64)
      INTEGER PREC

      DATA PREC/ 0,99, 0,99,98, 0, 0, 0, 0, 0,
     1           6, 3, 0, 0, 0,95, 0, 0, 3, 0,
     2           0, 0, 0,95, 0, 0,97,25, 0, 0,
     3           6, 0, 6,25, 6,30, 0, 0,60,60,
     4          70,70, 4, 3, 5, 0, 0, 0, 0,20,
     5          10, 0, 0, 1, 0, 0,98, 0,30,30,
     6          80,25,80,80                    /
      END
```

```
1         BLOCK DATA
2         COMMON/CLTAB/CLTAB(2,20),DSPCL(2,20)
3         COMMON/CLCAT/CLCAT(3,20)
4         INTEGER CLTAB,DSPCL,CLCAT
5
6         DATA CLTAB(1,1),DSPCL(1,1),DSPCL(1,2)/1,0,0/
7         DATA CLCAT(1,1),CLCAT(3,1),CLCAT(3,2)/25,0,0/
8         END
```

```
1       BLOCK DATA
2       COMMON/GLB/GLBNST(2,80),GLBTAB(80),IGLB,DSPGLB(20)
3       INTEGER GLBNST,GLBTAB,DSPGLB
4
5       DATA DSPGLB(1),IGLB/1,1/
6       END
```

```
      BLOCK DATA
      COMMON/CALADD/STADD(2,50),IADD,DSPADD(2,20)
      INTEGER STADD,DSPADD

      DATA DSPADD(1,1),DSPADD(1,2)/1,1/
      END
```

```
1         BLOCK DATA
2         COMMON/REN/RENTAB(2,100),IREN,DSPREN(5),NEWNAM(100)
3         INTEGER RENTAB,DSPREN
4
5         DATA NEWNAM(1),IREN,DSPREN(1)/65,2,1/
6         DATA RENTAB(1,1),RENTAB(2,1),RENTAB(1,2),RENTAB(2,2)/0,0,0,0/
7         END
```

```
1       BLOCK DATA
2       COMMON/PICPRG/DSPPIC(2,20),PICPRG(3,40),IPIC
3       INTEGER DSPPIC,PICPRG
4
5       DATA PICPRG(1,1),IPIC,DSPPIC(1,1)/1,1,1/
6       END
```

```
      BLOCK DATA
      COMMON/RESERV/RESERV


C LUCODE  ARITY  PRECEDENCE      OPER-SEPARATOR          RESERVED WORD
C ------  -----  ----------      --------------          -------------
C    0     -1        0
C    1      1       99          ALPHA
C    2     -1        0          BETA
C    3      2       99          GAMMA
C    4      2       98          LATEST
C    5     -1        0
C    6     -1        0
C    7     -1        0
C    8     -1        0
C    9     -1        0
C   10      2        6          AS SOON AS              ASA
C   11     -1        3          BEGIN OF EXPRESSION
C   12     -1        0          COMPUTE                 CMP(*)
C   13     -1        0          END                     END
C   14     -1        0
C   16     -1        0          FUNCTION                FUN(*)
C   17     -1        0
C   18      0        3          IF                      IF
C   19     -1        0
C   20     -1        0
C   15      1       95          FIRST                   FRST(*)
C   21     -1        0
C   22     -1        0          MAPPING                 MAP(*)
C   23      1       95          NEXT                    NXT(*)
C   24     -1        0          USING                   USG(*)
C   25     -1        0          PRODUCE                 PRO(*)
C   26      2       97          CONS
C   27      1       25          NOT                     NOT
C   28     -1        0          FALSE                   FALS(*)
C   29     -1        0          TRUE                    TRUE
C   30      2        6          UPON                    UPON
C   31     -1        0
C   32      2        5          WHENEVER                WHEN(*)
C   33      2       25          OR                      OR
C   34      2        6          FOLLOWED BY             FBY
C   35      2       30          EQ                      EQ
C   36     -1        0
C   37     -1        0
C   38      2       60          +
C   39      2       60          -
C   40      2       70          *
C   41      2       70          /
C   42      0        4          (
C   43      0        3          )
```

```
C  44        0         6
C  45       -1         0
C  46       -1         0
C  47       -1         0
C  48       -1         0
C  49        2        20              ELSE                    ELSE
C  50        2        10              CONDITIONAL             THEN
C  51       -1         0
C  52       -1         0              INV CHARACT
C  53        0         1              NONE END OF LINE
C  54       -1         0              NONE END OF FILE
C  55       -1         0              IDENTITY
C  56        2        98              CLAUSE APPLICATION
C  57       -1         0
C  58        2        30              <
C  59        2        30              >
C  60        1        80              E                       EXP
C  61        2        25              AND                     AND
C  62        2        80              **                      POW(*)
C  63        1        80              LN                      LN
C
C          NOTE : (*) MEANS THAT THE COMPLETE WORD CAN BE USED
      END
```

```
'BEGIN'
    'INTEGER' NOB,HGK;
    READ(NOB,HGK);

'BEGIN'
'REAL' RESULT;
'PROCEDURE' OUTPUT;'CODE';
'INTEGER' LCN,LLCN;
'INTEGER''ARRAY' OP(/1:HGK/),OPDE1(/1:HGK/),OPDE2(/1:HGK/),
                 LUCCAT(/1:NOB/);
'INTEGER''ARRAY' TIME(/1:10/); 'INTEGER'J;
'INTEGER''ARRAY' TREE(/1:300/); 'INTEGER' N;
'REAL''ARRAY' HBAC(/1:2,1:500/); 'INTEGER' L;
'INTEGER''ARRAY' CLCAT(/1:3,1:NOB/);
'INTEGER' I;
'REAL''PROCEDURE' EVAL(M,K);
  'INTEGER' M,K;
  'BEGIN'
  'PROCEDURE' ACTPAR(ACTADD,LL,M);
     'INTEGER' ACTADD,LL,M;
     'BEGIN'
          'INTEGER' T,D,I,J;
          T := TREE(/LCN+4/);
          LLCN := TREE(/LCN+1/);
          LL := TREE(/LLCN/);
          D := LUCCAT(/TREE(/LLCN+3/)/);
          J := OPDE1(/M/) - 1000 - 1;
          'IF' J=0 'THEN' T := OPDE2(/T/) + D - 1
                   'ELSE'
               'FOR' I := 1 'STEP' 1 'UNTIL' J 'DO'
                    T := OPDE2(/OPDE2(/T/)+D-1/);
          ACTADD := OPDE1(/T/) + D - 1;
          'IF' OP(/T/)¬=26 'THEN'
                 WRITE(''BAD NB OF ACTUAL PARA'');
     'END';
  'PROCEDURE' CLCALL(M,JJ,NAME,LSTNOD,LL,K,KK);
     'INTEGER' M,JJ,NAME,LSTNOD,LL,K,KK;
     'BEGIN'
          'INTEGER' T;
          T := TREE(/LCN+3/);
          LL := LUCCAT(/T+1/) - LUCCAT(/T/);
          T := CLCAT(/2,T/);
          T := M - T + 1;
          T := LCN + 4 + T;
          'IF' TREE(/T/)>0 'THEN'
               'BEGIN' LLCN := TREE(/T/);
                       LL := TREE(/LLCN/);
                       'GO TO' TIM;
               'END';
          TREE(/T/) := N;
```

```
                LL := LL + TREE(/LCN/);
                T := LL + LUCCAT(/NAME+1/) - LUCCAT(/NAME/);
                'IF' T>500 'THEN'
                     'BEGIN' WRITE(''INV INDEX: HBAC OVF'');
                             'GO TO' EXIT;
                     'END';
                TREE(/N/) := LL;
                TREE(/N+1/) := LCN;
                TREE(/N+2/) := LSTNOD;
                TREE(/N+3/) := NAME;
                TREE(/N+4/) := M;
                LLCN := N;
                N := N + 5 + CLCAT(/3,NAME/);
                'IF' N>494 'THEN'
                     'BEGIN' WRITE(''INV INDEX: TREE OVF'');
                             'GO TO' EXIT;
                     'END';
TIM:            KK := K;
                JJ := J;
                'IF' CLCAT(/1,NAME/)¬=12'AND'CLCAT(/1,NAME/)¬=22 'THEN'
                     'GO TO' EXIT;
                JJ := J + 1;
                'IF' JJ>10 'THEN'
                     'BEGIN' WRITE(''INV INDEX: TIME OVF'');
                             'GO TO' EXIT;
                     'END';
                TIME(/JJ/) := K;
                KK := 0;
EXIT:      'END';
           'PROCEDURE' GLOBAL(LSTNOD,A,LL);
              'INTEGER' LSTNOD,A,LL;
              'BEGIN'
                   'INTEGER' I;
                   LSTNOD := LCN;
                   'FOR' I := 1 'STEP' 1 'UNTIL' A - 1000 'DO'
                        LSTNOD := TREE(/LSTNOD+2/);
                   LL := TREE(/LSTNOD/);
              'END';
           'INTEGER' OPCODE,SAVTIM,SAVNOD,SAVL,LL,LSTNOD,NAME,KK,JJ,DD,II,
                     MM,ACTADD,D,I1,I2,I3;
           'REAL' T1,T2;
           'SWITCH' AIG :=      ALFA,BETA,GAMA,LTST,OPER,OPER,OPER,OPER,OPER,
                          ASA,OPER,OPER,OPER,OPER, FST,OPER,OPER,OPER,OPER,
                         OPER,OPER,OPER, NXT,OPER,OPER,OPER, NOT,OPER,OPER,
                         UPON,OPER,WHEN,  OR, FBY,  EQ,OPER,OPER, ADD, SUB,
                          MUL, DIV,OPER,OPER,OPER,OPER,OPER,OPER,OPER,OPER,
                         COND,OPER,OPER,OPER,OPER,IDEN,OPER,OPER,  LT,  GT,
                         POW , AND,FEXP, FLN;

           OPCODE := OP(/M/);
```

```
        'GO TO' AIG(/OPCODE/);
OPER:   OUTPUT(1,'('('*****  INVALID OPCODE')'2(2D)')',OPCODE,M);

ALFA:   D := LUCCAT(/TREE(/LCN+3/)/);
        DD := M - D + L;
        'IF' HBAC(/1,DD/)=K 'THEN'
           'BEGIN' EVAL := HBAC(/2,DD/);
                   'GO TO' EXIT;
           'END';
        SAVNOD := LCN;
        SAVL := L;
        ACTPAR(ACTADD,LL,M);
        LCN := LLCN;
        L := LL;
        D := LUCCAT(/TREE(/LCN+3/)/);
        'IF' OPDE1(/ACTADD/)>=500 'THEN'
           T1 := 'IF' OPDE1(/ACTADD/)>=1000 'THEN' OPDE1(/ACTADD/) - 1000
                                            'ELSE' OPDE1(/ACTADD/) - 500
                                 'ELSE' T1 := EVAL(ACTADD,K);
        HBAC(/1,DD/) := K;
        HBAC(/2,DD/) := T1;
        EVAL := T1;
        L := SAVL;
        LCN := SAVNOD;
        'GO TO' EXIT;

BETA:   D := LUCCAT(/TREE(/LCN+3/)/);
        DD := M - D + L;
        'IF' HBAC(/1,DD/)=K 'THEN'
           'BEGIN' EVAL := HBAC(/2,DD/);
                   'GO TO' EXIT;
           'END';
        SAVNOD := LCN;
        SAVL := L;
        SAVTIM := J;
        'IF' OPDE1(/M/)>=500 'THEN'
           'BEGIN' NAME := OPDE1(/M/) - 500;
                   LSTNOD := LCN;
           'END'
                              'ELSE'
           'BEGIN' NAME := ENTIER(EVAL(OPDE1(/M/)+D-1,K));
                   LSTNOD := 'IF' LLCN=1 'THEN' 1
                                         'ELSE' TREE(/LLCN+2/);
           'END';
        CLCALL(M,JJ,NAME,LSTNOD,LL,K,KK);
        J := JJ;
        LCN := LLCN;
        L := LL;
        D := LUCCAT(/TREE(/LCN+3/)/);
        T1 := EVAL(D,KK);
```

```
        LCN := SAVNOD;
        L := SAVL;
        J := SAVTIM;
        HBAC(/1,DD/) := K;
        HBAC(/2,DD/) := T1;
        EVAL := T1;
        'GO TO' EXIT;

GAMA:   'IF' OPDE2(/M/)>=500 'THEN'
           'BEGIN' GLOBAL(LSTNOD,OPDE1(/M/),LL);
                   LLCN := LSTNOD;
                   EVAL := OPDE2(/M/) - 500;
                   'GO TO' EXIT;
           'END';
        D := LUCCAT(/TREE(/LCN+3/)/);
        DD := M - D + L;
        'IF' HBAC(/1,DD/)=K 'THEN'
           'BEGIN' EVAL := HBAC(/2,DD/);
                   'GO TO' EXIT;
           'END';
        SAVNOD := LCN;
        SAVL := L;
        GLOBAL(LSTNOD,OPDE1(/M/),LL);
        LCN := LSTNOD;
        L := LL;
        D := LUCCAT(/TREE(/LCN+3/)/);
        T1 := EVAL(OPDE2(/M/)+D-1,K);
        LCN := SAVNOD;
        L := SAVL;
        HBAC(/1,DD/) := K;
        HBAC(/2,DD/) := T1;
        EVAL := T1;
        'GO TO' EXIT;

LTST:   D := LUCCAT(/TREE(/LCN+3/)/);
        DD := M - D + L;
        'IF' HBAC(/1,DD/)=K 'THEN'
           'BEGIN' EVAL := HBAC(/2,DD/);
                   'GO TO' EXIT;
           'END';
        T1 := EVAL(OPDE2(/M/)+D-1,TIME(/J-(OPDE1(/M/)-1000)/));
        HBAC(/1,DD/) := K;
        HBAC(/2,DD/) := T1;
        EVAL := T1;
        'GO TO' EXIT;

ASA:    D := LUCCAT(/TREE(/LCN+3/)/);
        DD := M - D + L;
        'IF' HBAC(/1,DD/)=K 'THEN'
           'BEGIN' EVAL := HBAC(/2,DD/);
```

```
                'GO TO' EXIT;
        'END';
      MM := OPDE2(/M/) + D - 1;
      II := -1;
      'IF' OPDE2(/M/)>=1000 'THEN'
        'BEGIN' T1 := OPDE2(/M/) - 1000;
LOOP:           II := II + 1;
                'IF' T1=1 'THEN' 'GO TO' TRUE;
                WRITE(' LOOP IN ASA EXP');
                'GO TO' LOOP;
        'END';
      'FOR' I := 0,I+1 'WHILE' EVAL(MM,I)=0 'DO'
        II := I;
TRUE: T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                  'ELSE' EVAL(OPDE1(/M/)+D-1,II);
      HBAC(/1,DD/) := K;
      HBAC(/2,DD/) := T1;
      EVAL := T1;
      'GO TO' EXIT;

FST:  D := LUCCAT(/TREE(/LCN+3/)/);
      DD := M - D + L;
      'IF' HBAC(/1,DD/)=K 'THEN'
        'BEGIN' EVAL := HBAC(/2,DD/);
                'GO TO' EXIT;
        'END';
      T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                  'ELSE' EVAL(OPDE1(/M/)+D-1,0);
      HBAC(/1,DD/) := K;
      HBAC(/2,DD/) := T1;
      EVAL := T ;
      'GO TO' EXIT;

NXT:  D := LUCCAT(/TREE(/LCN+3/)/);
      DD := M - D + L;
      'IF' HBAC(/1,DD/)=K 'THEN'
        'BEGIN' EVAL := HBAC(/2,DD/);
                'GO TO' EXIT;
        'END';
      T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                  'ELSE' EVAL(OPDE1(/M/)+D-1,K+1);
      HBAC(/1,DD/) := K;
      HBAC(/2,DD/) := T1;
      EVAL := TF;
      'GO TO' EXIT;

NOT:  D := LUCCAT(/TREE(/LCN+3/)/);
      DD := M - D + L;
      'IF' HBAC(/1,DD/)=K 'THEN'
        'BEGIN' EVAL := HBAC(/2,DD/);
```

```
                'GO TO' EXIT;
        'END';
    T1 := 'IF' OPDE1(/M/)>=1000 'THEN' ABS(OPDE1(/M/)-1000-1)
                                'ELSE' ABS(EVAL(OPDE1(/M/)+D-1,K)-1);
    HBAC(/1,DD/) := K;
    HBAC(/2,DD/) := T1;
    EVAL := T1;
    'GO TO' EXIT;

UPON: D := LUCCAT(/TREE(/LCN+3/)/);
    DD := M - D + L;
    'IF' HBAC(/1,DD/)=K 'THEN'
        'BEGIN' EVAL := HBAC(/2,DD/);
                'GO TO' EXIT;
        'END';
    'IF' K=0 'THEN' 'BEGIN' T1 := EVAL(OPDE1(/M/)+D-1,0);
                            'GO TO' OUTU;
                    'END';
    'IF' EVAL(OPDE2(/M/)+D-1,K-1)¬=1 'THEN'
        'BEGIN' T1 := EVAL(M,K-1);
                'GO TO' OUTU;
        'END';
    I1 := I2 := 0;
    I3 := OPDE2(/M/) + D - 1;
UP: 'IF' EVAL(I3,I1)=1 'THEN' I2 := I2 + 1;
    I1 := I1 + 1;
    'IF' I1=K 'THEN' 'BEGIN' T1 := EVAL(OPDE1(/M/)+D-1,I2);
                             'GO TO' OUTU;
                     'END';
    'GO TO' UP;
OUTU: HBAC(/1,DD/) := K;
    HBAC(/2,DD/) := T1;
    EVAL := T1;
    'GO TO' EXIT;

WHEN: D := LUCCAT(/TREE(/LCN+3/)/);
    DD := M - D + L;
    'IF' HBAC(/1,DD/)=K 'THEN'
        'BEGIN' EVAL := HBAC(/2,DD/);
                'GO TO' EXIT;
        'END';
    I1 := I2 := 0;
    I3 := OPDE2(/M/) + D - 1;
BACK: 'IF' EVAL(I3,I1)=1 'THEN'
        'BEGIN' 'IF' I2=K 'THEN' 'GO TO' OUTW;
                I2 := I2 + 1;
        'END';
    I1 := I1 + 1;
    'GO TO' BACK;
OUTW: T1 := EVAL(OPDE1(/M/)+D-1,I1);
```

```
      HBAC(/1,DD/) := K;
      HBAC(/2,DD/) := T1;
      EVAL := T1;
      'GO TO' EXIT;

OR:   D := LUCCAT(/TREE(/LCN+3/)/);
      DD := M - D + L;
      'IF' HBAC(/1,DD/)=K 'THEN'
         'BEGIN' EVAL := HBAC(/2,DD/);
                 'GO TO' EXIT;
         'END';
      T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                  'ELSE' EVAL(OPDE1(/M/)+D-1,K);
      'IF' T1=1 'THEN' 'GO TO' RES;
      T1 := 'IF' OPDE2(/M/)>=1000 'THEN' OPDE2(/M/) - 1000
                                  'ELSE' EVAL(OPDE2(/M/)+D-1,K);
RES:  HBAC(/1,DD/) := K;
      HBAC(/2,DD/) := T1;
      EVAL := T1;
      'GO TO' EXIT;

FBY:  D := LUCCAT(/TREE(/LCN+3/)/);
      DD := M - D + L;
      'IF' HBAC(/1,DD/)=K 'THEN'
         'BEGIN' EVAL := HBAC(/2,DD/);
                 'GO TO' EXIT;
         'END';
      'IF' K=0 'THEN'
         T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                     'ELSE' EVAL(OPDE1(/M/)+D-1,0)
               'ELSE'
         T1 := 'IF' OPDE2(/M/)>=1000 'THEN' OPDE2(/M/) - 1000
                                     'ELSE' EVAL(OPDE2(/M/)+D-1,K-1);
      HBAC(/1,DD/) := K;
      HBAC(/2,DD/) := T1;
      EVAL := T1;
      'GO TO' EXIT;

EQ:   D := LUCCAT(/TREE(/LCN+3/)/);
      DD := M - D + L;
      'IF' HBAC(/1,DD/)=K 'THEN'
         'BEGIN' EVAL := HBAC(/2,DD/);
                 'GO TO' EXIT;
         'END';
      T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                  'ELSE' EVAL(OPDE1(/M/)+D-1,K);
      T2 := 'IF' OPDE2(/M/)>=1000 'THEN' OPDE2(/M/) - 1000
                                  'ELSE' EVAL(OPDE2(/M/)+D-1,K);
      T1 := 'IF' T1=T2 'THEN' 1
                       'ELSE' 0;
```

```
      HBAC(/1,DD/) := K;
      HBAC(/2,DD/) := T1;
      EVAL := T1;
      'GO TO' EXIT;

ADD:  D := LUCCAT(/TREE(/LCN+3/)/);
      DD := M - D + L;
      'IF' HBAC(/1,DD/)=K 'THEN'
         'BEGIN' EVAL := HBAC(/2,DD/);
                 'GO TO' EXIT;
         'END';
      T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                  'ELSE' EVAL(OPDE1(/M/)+D-1,K);
      T2 := 'IF' OPDE2(/M/)>=1000 'THEN' OPDE2(/M/) - 1000
                                  'ELSE' EVAL(OPDE2(/M/)+D-1,K);
      T1 := T1 + T2;
      HBAC(/1,DD/) := K;
      HBAC(/2,DD/) := T1;
      EVAL := T ;
      'GO TO' EXIT;

SUB:  D := LUCCAT(/TREE(/LCN+3/)/);
      DD := M - D + L;
      'IF' HBAC(/1,DD/)=K 'THEN'
         'BEGIN' EVAL := HBAC(/2,DD/);
                 'GO TO' EXIT;
         'END';
      T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                  'ELSE' EVAL(OPDE1(/M/)+D-1,K);
      T2 := 'IF' OPDE2(/M/)>=1000 'THEN' OPDE2(/M/) - 1000
                                  'ELSE' EVAL(OPDE2(/M/)+D-1,K);
      T1 := T1 - T2;
      HBAC(/1,DD/) := K;
      HBAC(/2,DD/) := T1;
      EVAL := T1;
      'GO TO' EXIT;

MUL:  D := LUCCAT(/TREE(/LCN+3/)/);
      DD := M - D + L;
      'IF' HBAC(/1,DD/)=K 'THEN'
         'BEGIN' EVAL := HBAC(/2,DD/);
                 'GO TO' EXIT;
         'END';
      T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                  'ELSE' EVAL(OPDE1(/M/)+D-1,K);
      T2 := 'IF' OPDE2(/M/)>=1000 'THEN' OPDE2(/M/) - 1000
                                  'ELSE' EVAL(OPDE2(/M/)+D-1,K);

      T1 := T1 * T2;
      HBAC(/1,DD/) := K;
      HBAC(/2,DD/) := T1;
```

```
        EVAL := TF;
        'GO TO' EXIT;

DIV:    D := LUCCAT(/TREE(/LCN+3/)/);
        DD := M - D + L;
        'IF' HBAC(/1,DD/)=K 'THEN'
           'BEGIN' EVAL := HBAC(/2,DD/);
                   'GO TO' EXIT;
           'END';
        T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                    'ELSE' EVAL(OPDE1(/M/)+D-1,K);
        T2 := 'IF' OPDE2(/M/)>=1000 'THEN' OPDE2(/M/) - 1000
                                    'ELSE' EVAL(OPDE2(/M/)+D-1,K);
        T1 := T1 / T2;
        HBAC(/1,DD/) := K;
        HBAC(/2,DD/) := T1;
        EVAL := T1;
        'GO TO' EXIT;

COND:   D := LUCCAT(/TREE(/LCN+3/)/);
        DD := M - D + L;
        'IF' HBAC(/1,DD/)=K 'THEN'
           'BEGIN' EVAL := HBAC(/2,DD/);
                   'GO TO' EXIT;
           'END';
        T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                    'ELSE' EVAL(OPDE1(/M/)+D-1,K);
        T2 := OPDE2(/M/) + D - 1;
        'IF' T1=1 'THEN'
           'BEGIN' T1 := 'IF' OPDE1(/T2/)>=1000 'THEN' OPDE1(/T2/) - 1000
                                                'ELSE' EVAL(OPDE1(/T2/)+D-1,K);
           'END'
                 'ELSE'
           'BEGIN' T1 := 'IF' OPDE2(/T2/)>=1000 'THEN' OPDE2(/T2/) - 1000
                                                'ELSE' EVAL(OPDE2(/T2/)+D-1,K);
           'END';
        HBAC(/1,DD/) := K;
        HBAC(/2,DD/) := T1;
        EVAL := T1;
        'GO TO' EXIT;

IDEN:   D := LUCCAT(/TREE(/LCN+3/)/);
        DD := M - D + L;
        'IF' HBAC(/1,DD/)=K 'THEN'
           'BEGIN' EVAL := HBAC(/2,DD/);
                   'GO TO' EXIT;
           'END';
        T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                    'ELSE' EVAL(OPDE1(/M/)+D-1,K);
        HBAC(/1,DD/) := K;
```

```
      HBAC(/2,DD/) := T1;
      EVAL := T1;
      'GO TO' EXIT;

LT:   D := LUCCAT(/TREE(/LCN+3/)/);
      DD := M -D + L;
      'IF' HBAC(/1,DD/)=K 'THEN'
         'BEGIN' EVAL := HBAC(/2,DD/);
                 'GO TO' EXIT;
         'END';
      T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                  'ELSE' EVAL(OPDE1(/M/)+D-1,K);
      T2 := 'IF' OPDE2(/M/)>=1000 'THEN' OPDE2(/M/) - 1000
                                  'ELSE' EVAL(OPDE2(/M/)+D-1,K);
      T1 := 'IF' T1<T2 'THEN' 1
                       'ELSE' 0;
      HBAC(/1,DD/) := K;
      HBAC(/2,DD/) := T1;
      EVAL := T ;
      'GO TO' EXIT;

GT:   D := LUCCAT(/TREE(/LCN+3/)/);
      DD := M -D + L;
      'IF' HBAC(/1,DD/)=K 'THEN'
         'BEGIN' EVAL := HBAC(/2,DD/);
                 'GO TO' EXIT;
         'END';
      T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                  'ELSE' EVAL(OPDE1(/M/)+D-1,K);
      T2 := 'IF' OPDE2(/M/)>=1000 'THEN' OPDE2(/M/) - 1000
                                  'ELSE' EVAL(OPDE2(/M/)+D-1,K);
      T1 := 'IF' T1>T2 'THEN' 1
                       'ELSE' 0;
      HBAC(/1,DD/) := K;
      HBAC(/2,DD/) := T1;
      EVAL := T1;
      'GO TO' EXIT;

FEXP: D := LUCCAT(/TREE(/LCN+3/)/);
      DD := M - D + L;
      'IF' HBAC(/1,DD/)=K 'THEN'
         'BEGIN' EVAL := HBAC(/2,DD/);
                 'GO TO' EXIT;
         'END';
      T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                  'ELSE' EVAL(OPDE1(/M/)+D-1,K);
      T1 := EXP(T1);
      HBAC(/1,DD/) := K;
      HBAC(/2,DD/) := T1;
      EVAL := T1;
```

```
      'GO TO' EXIT;

POW:  D := LUCCAT(/TREE(/LCN+3/)/);
      DD := M -D + L;
      'IF' HBAC(/1,DD/)=K 'THEN'
         'BEGIN' EVAL := HBAC(/2,DD/);
                 'GO TO' EXIT;
         'END';
      T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                  'ELSE' EVAL(OPDE1(/M/)+D-1,K);
      T2 := 'IF' OPDE2(/M/)>=1000 'THEN' OPDE2(/M/) - 1000
                                  'ELSE' EVAL(OPDE2(/M/)+D-1,K);
      T1 := T1 ** T2;
      HBAC(/1,DD/) := K;
      HBAC(/2,DD/) := T1;
      EVAL := T1;
      'GO TO' EXIT;

AND:  D := LUCCAT(/TREE(/LCN+3/)/);
      DD := M -D + L;
      'IF' HBAC(/1,DD/)=K 'THEN'
         'BEGIN' EVAL := HBAC(/2,DD/);
                 'GO TO' EXIT;
         'END';
      T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                  'ELSE' EVAL(OPDE1(/M/)+D-1,K);
      T2 := 'IF' OPDE2(/M/)>=1000 'THEN' OPDE2(/M/) - 1000
                                  'ELSE' EVAL(OPDE2(/M/)+D-1,K);
      T1 := 'IF' T1=1&T2=1 'THEN' 1
                           'ELSE' 0;
      HBAC(/1,DD/) := K;
      HBAC(/2,DD/) := T1;
      EVAL := T1;
      'GO TO' EXIT;

FLN:  D := LUCCAT(/TREE(/LCN+3/)/);
      DD := M - D + L;
      'IF' HBAC(/1,DD/)=K 'THEN'
         'BEGIN' EVAL := HBAC(/2,DD/);
                 'GO TO' EXIT;
         'END';
      T1 := 'IF' OPDE1(/M/)>=1000 'THEN' OPDE1(/M/) - 1000
                                  'ELSE' EVAL(OPDE1(/M/)+D-1,K);
      T1 := LN(T1);
      HBAC(/1,DD/) := K;
      HBAC(/2,DD/) := T1;
      EVAL := T1;
      'GO TO' EXIT;

EXIT: 'END';
```

```
                LCN := 1;
                INTARRAY(3,OP);
                INTARRAY(3,OPDE1);
                INTARRAY(3,OPDE2);
                INTARRAY(3,LUCCAT);
                INTARRAY(3,CLCAT);
                'FOR' I := 1 'STEP' 1 'UNTIL' 300 'DO'
                  'BEGIN' HBAC(/1,I/) := -1;
                          TREE(/I/) := 0;
                  'END';
                'FOR' I := 301 'STEP' 1 'UNTIL' 500 'DO'
                  HBAC(/1,I/) := -1;
                N := 5 + CLCAT(/3,1/) + 1;
                TREE(/1/) := TREE(/4/) := L := 1;
                J := 0;
      'FOR' I := 0 'STEP' 1 'UNTIL' 9 'DO'
         'BEGIN' RESULT := EVAL(1,I);
                 PRINT (RESULT);
         'END';
   'END';
'END'
```